



CD-ROM
INCLUDED

bject **REXX** for OS/2 Warp

- ◆ Take the “oh, oh!” out of OO
- ◆ Includes Object REXX for OS/2
- ◆ Object Extensions for Procedural REXX
- ◆ Access to DB2/2 Version 2 Large Objects
- ◆ Create Internet Web Pages

IBM

**Trevor Turton
Ueli Wahli**



International Technical Support Organization

Object REXX for OS/2 Warp

Object REXX for OS/2 Warp

Trevor Turton
Ueli Wahli



INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION
SAN JOSE, CALIFORNIA 95120

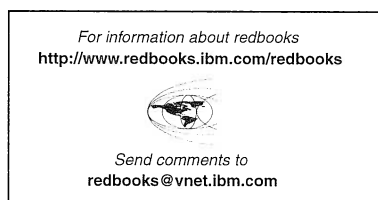
PRENTICE HALL PTR
UPPER SADDLE RIVER, NEW JERSEY 07458
[HTTP://WWW.PRENHALL.COM](http://www.prenhall.com)

© Copyright International Business Machines Corporation 1997. All rights reserved.

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

This edition applies to Object REXX running under OS/2 Warp.

Comments about ITSO Technical Bulletins may be addressed to:
IBM Corporation ITSO, Almaden Research Center, 471/80-E2, 650 Harry Road, San Jose, California 95120-6099



Published by Prentice Hall PTR
Prentice-Hall, Inc.
A Simon & Schuster Company
Upper Saddle River, NJ 07458

Acquisitions Editor: Michael E. Meehan

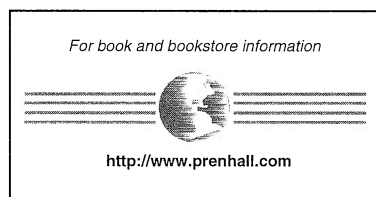
Manufacturing Manager: Alexis R. Heydt

Cover Design: Ueli Wahli, Ingo Holder, Design Source

Copy Editor: Maggie Cutler, Marti Jones

Editorial/Production Supervision: Joe Czerwinski

The publisher offers discounts on this book when ordered in bulk quantities. For more information, contact:
Corporate Sales Department, Prentice Hall PTR, One Lake Street, Upper Saddle River, NJ 07458
Phone: 800-382-3419; FAX: 201-236-7141; E-mail (Internet): corpsales@prenhall.com



Printed in the United States of America

1 0 9 8 7 6 5 4 3 2 1

ISBN 0-13-273467-2

Prentice-Hall International (UK) Limited, *London*

Prentice-Hall of Australia Pty. Limited, *Sydney*

Prentice-Hall Canada Inc., *Toronto*

Prentice-Hall Hispanoamericana, S.A., *Mexico*

Prentice-Hall of India Private Limited, *New Delhi*

Prentice-Hall of Japan, Inc., *Tokyo*

Simon & Schuster Asia Pte. Ltd., *Singapore*

Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Abstract

Object orientation (OO) is a topic of great interest and concern today. Some developers who use OO claim that it significantly increases productivity; others view it as good for rocket science, not for business.

Many OO languages seem complicated and alien to programmers familiar with procedural languages such as COBOL. This book introduces Object REXX, a new OO language that breaks the OO barrier. Object REXX is based on a tried-and-trusted language used around the world today. Because it has the most complete and easy-to-use set of OO features of any language, it offers a simple way for programmers with a procedural background to enter the new world of objects. Object REXX also supports distributed objects written in many other languages through Common Object Request Broker Architecture (CORBA) technologies, such as IBM's System Object Model (SOM).

This book demonstrates a practical approach to using Object REXX and OO techniques to develop commercial systems to meet changing business requirements. It tells the story of how Hanna, Steve, and Curt design and implement a commercial application system step by step, using object persistence in file systems and relational databases, graphical user interface (GUI) builders, existing SOM-based objects, the OS/2 Workplace Shell (WPS), and Internet Web pages. Extensive code examples are provided to illustrate every step.

To Hanna, Curt, and Steve, the hard-working people at the fictitious Hacurs company. Originally loosely defined, they inspired the writing of real-life dialog to portray a small company trying to find a niche in the marketplace. Any resemblance to real people is completely coincidental.

To my wife Ingrid, for the love and patience that she showed while her husband labored over recalcitrant programs and screeds of text.

Trevor

To my wife, Patricia, for her ongoing support and understanding of long hours at work. To the staff at the ITSO San Jose center for making work a joy, at least most of the time.

Ueli

Contents

Abstract	v
Figures	xv
Tables	xix
Special Notices	xxi
Preface	xxiii
How This Document Is Organized	xxiv
Related Publications	xxvii
International Technical Support Organization Publications	xxviii
ITSO Redbooks on the World Wide Web (WWW)	xxix
ITSO Redbooks and Sample Code on the Internet	xxix
About the Authors	xxx
Acknowledgments	xxx

Part 1. Introducing Object REXX 1

Chapter 1. Introducing Object REXX	3
What's New in Object REXX?	3
Why REXX?	5
Why Object Orientation?	6
The Productivity Problem	6
The Reuse Solution	7
The Waterfall Method	8
The Spiral Method	8
Prototyping	9
The Paradigm Shift	9
Better Reuse from the OO Approach	12
Communities of Cooperative Objects	13
Bloated PC Software	13
Standard Software Components	14
Liberating Objects from Applications	14
The CORBA Standard	14
So Why Object REXX?	15
Chapter 2. How Does Object REXX Implement OO?	17
Objects	18
Classes	19
Inheritance	20
Abstract Classes	21
Multiple Inheritance	23
Object REXX Variable Pools	25
Object Instances	27

Object Creation	27
Object Destruction	28
Methods	28
Private and Public Methods	29
Classes and Instance Methods	29
Meta Classes	30
Polymorphism	30
The Object REXX Class Library	33
The Object REXX Class Library Browser	34

Part 2. The Car Dealer Scenario 35

Chapter 3. The Car Dealer Application 37

Introducing the Hacurs Company	37
The Car Dealer Opportunity	38
The Application Model	41
Methods and Variables	43
Relationships Among Objects	48
The Object REXX Collection Classes	49
Object Creation and Destruction	50
Implementation of the Model in Memory	52
Implementation Notes	53
Sample Class Definition	54
Source Code for Base Class Implementation	55

Chapter 4. ASCII User Interface 57

Designing the User Interface	57
ASCII User Interface As an Object	61
The AUI Class	62
The AUI Operations	62
ASCII Menus as Objects	63
The Menu Operations	63
Implementing the Menus	64
Appearance of ASCII User Interface	65
Source Code for ASCII User Interface	65

Chapter 5. Persistent Objects on Disk 67

Storing Objects in FAT Files	67
Format of the Objects	71
Implementing the Changes in Code	72
The Class Structure	74
The Requires Directive	76
The Persistent Class	77
Source Code and Sample Data for FAT Class Implementation	79

Chapter 6. Graphical User Interfaces 81

The Setup	81
The Car Dealer GUI	82

Choice of GUI Builders	88
How to Include Directives in GUI Builders	92
Directives in Dr. Dialog	92
Directives in VisPro/REXX	93
Directives in Watcom VX•REXX	94
GUI Builder Development Environment	95
Development Environment for Dr. Dialog	95
Development Environment for VisPro/REXX	97
Development Environment for Watcom VX•REXX	99
Testing and Generating the GUI Applications	100
Chapter 7. Persistent Objects in DB2	101
Storing Objects in DB2	101
Persistent Methods for DB2 Support	106
Implementation of DB2 Support	108
Implementation of Load at Application Start	109
Implementation of Load-on-Demand	109
Implementation Notes	110
Source Code for DB2 Class Implementation	110
Chapter 8. Using Advanced DB2/2 Facilities	111
Multimedia in DB2 BLOBs	111
Using DB2 BLOBs from Object REXX	114
Multiple Multimedia Files in BLOBs	118
Implementing the DB2 Multimedia Support	125
Implementation Notes	128
Source Code for DB2 Multimedia Implementation	128
Chapter 9. Data Security with Object REXX and DB2	129
The Security Problem	129
Coding Stored Procedures with Object REXX	133
Chapter 10. Configuration Management with Object REXX	143
Breaking an Application into Multiple Files	144
Using Multiple Subdirectories	147
Controlling Which Files Are Used	148
Overall Car Dealer File Structure	150
Communication Among Classes	152
The Local Directory	153
The Global Directory	154
Installation Program Considerations	155
Implementation of Configuration Files	157
Using the Configuration File	158
Configuration File for List Routines	159
Implementation of the Car Dealer Class	159
Using the Car Dealer Class	160
Source Code for Configuration Management	160
Chapter 11. Object REXX, SOM, and Workplace Shell	161
Using SOM in the Car Dealer Application	162
Hacurs Builds a SOM Object	163

How the SOM Object Was Implemented	166
Implementation Steps	170
Running the Application with the SOM Part	172
Implementation Notes	172
Source Code for SOM Implementation	172
Object REXX and the OS/2 Workplace Shell	173
Car Dealer Data in the Workplace Shell	174
Implementation Notes	178
Source Code	178
Applications Assembled from Components	179
Chapter 12. Object REXX and the World Wide Web	183
Hacurs Connects to the Internet	184
Hacurs Makes a Plan for the Web	184
Hacurs Designs a Home Page	187
The Home Page	187
Web Car Dealer Application	191
Web Common Gateway Interface	192
HTML Class	194
Customer Search Form	197
Program Organization and Performance	200
Customizing the File Organization on the Web Server	201
Optimizing Performance	203
Car Dealer Start Program for the Web	203
Car Dealer Common Interface Program	204
Multimedia on the Web	206
Interacting with Web Users	208
Adding a Web Customer	209
Car Dealer Home Page	212
Implementation Notes	214
Source Code	214
Part 3. Object REXX and Concurrency	215
Chapter 13. Object REXX and Concurrency	217
Object-Based Concurrency	217
The Object REXX Concurrency Facilities	218
Early Reply	218
Message Objects	218
Unguarded Methods	219
The Guard Instruction	220
Examples of Early Reply with Unguarded and Guarded Methods	220
Philosopher's Forks	223
Philosopher's Forks in an OS/2 Window	223
Visualizing Philosopher's Forks with a GUI	227
GUI Design of the Philosopher's Forks with Dr. Dialog	230
GUI Design of the Philosopher's Forks with VisPro/REXX	235
GUI Design of the Philosopher's Forks with Watcom VX•REXX	236

Part 4. Installing Object REXX and the Sample Applications 239

Chapter 14. Installing Object REXX and the Sample Applications	241
Content of the CD	241
Installation of Object REXX	242
Running the Sample Applications from the CD	242
Installing the Sample Applications	242
Prerequisites for the Sample Applications	243
Installation Program	243
Installation of the Code	245
Installing the Car Dealer and Philosopher's Forks Applications	245
Update of Config.Sys	245
Create the ObjectRexx Redbook Folder	246
DB2 Setup	250
Define the DB2 Database	250
Define the DB2 Tables	251
Load the DB2 Tables	252
Running the Sample Applications	253
Running the Car Dealer Application on the World Wide Web	254
Installed Sample Applications	254
Car Dealer Directory	254
Philosopher's Forks Directory	260
Source Code for Running Sample Applications	260
Removing the Sample Applications from Your System	260

Part 5. New Features and Syntax in Object REXX 261

Chapter 15. New Features in Object REXX and Migration	263
Object-Oriented Facilities	264
New Special Variables	264
Special and Built-In Objects	265
Directives	265
Class Directive	265
Method Directive	266
Routine Directive	267
Requires Directive	267
The REXXC Utility	267
New and Enhanced Instructions	268
CALL (Enhanced)	268
DO (Enhanced)	269
EXPOSE (New)	270
FORWARD (New)	270
GUARD (New)	271
PARSE (Enhanced)	271
RAISE (New)	272
REPLY (New)	273
SIGNAL (Enhanced)	274

USE (New)	275
New and Enhanced Built-In Functions	275
ARG (Enhanced)	275
CHANGESTR (New)	275
CONDITION (Enhanced)	276
COUNTSTR (New)	276
DATATYPE (Enhanced)	276
DATE (Enhanced)	277
STREAM (Enhanced)	277
TIME (Enhanced)	279
VAR (New)	279
New Condition Traps	280
CALL/SIGNAL (Enhanced)	280
New REXX Utilities	282
Utilities for WPS	282
Utilities or Semaphores	282
Utilities for REXX Macros	283
Utilities for Files	284
Utilities for Code Pages	284
Utilities for OS/2 Systems	285
Migration Considerations	286
Appendix A. Car Dealer Source Code	287
Directory Structure	288
Car Dealer Application	288
Philosopher's Forks	289
Car Dealer Source Code	289
Sample Data	289
Sample Customer Data	289
Sample Vehicle Data	290
Sample Work Order Data	290
Sample Service Item Data	291
Sample Part Data	291
Multimedia Setup	292
Multimedia Data Definition File	292
Base Classes	293
Base Customer Class	293
Base Vehicle Class	296
Base Work Order Class	297
Base Service Item Class	302
Base Part Class	304
Base Part Class as Subclass of a SOM Class	307
Persistent Class	308
Cardeal Class	308
Persistence in Files	310
Configuration for File Storage	310
File Customer Class	311
File Vehicle Class	312
File Work Order Class	314
File Service Item Class	315

File Part Class	316
Persistence in DB2	317
Configuration for DB2 Storage	317
DB2 Customer Class	317
DB2 Vehicle Class	319
DB2 Work Order Class	322
DB2 Service Item Class	325
DB2 Part Class	326
Running the Car Dealer Application	327
Program to Run the Car Dealer Application	327
Appendix B. Definition for Syntax Diagram Structure	329
Index	331

Figures

1. Car and DumpTruck Class Inheritance Diagram	20
2. Abstract Class Inheritance Diagram	21
3. Multilevel Class Inheritance Diagram	22
4. Mixin Class Multiple Inheritance Diagram	24
5. Car Dealer Application Use Case	39
6. Car Dealer Data Class Relationships	41
7. Car Dealer Object Attributes	42
8. Implementation of the Car Dealer Model	52
9. Customer Class in Memory	54
10. Appearance of ASCII User Interface	66
11. Customer Class Inheritance Diagram	73
12. FAT Data Classes Inheriting from a Mixin Class	78
13. Main Window of Dr. Dialog GUI Application	83
14. Part List Window of Dr. Dialog GUI Application	84
15. Service Items List Window of Dr. Dialog GUI Application	85
16. Work Orders Window of Dr. Dialog GUI Application	86
17. Billing Window of Dr. Dialog GUI Application	87
18. Dr. Dialog Project Folder	95
19. Dr. Dialog Development Environment: Window Layout	95
20. Dr. Dialog Development Environment: DrRexx Notebook	96
21. VisPro/REXX Project Folder	97
22. VisPro/REXX Development Environment: Layout View	97
23. VisPro/REXX Development Environment: Event Tree View	98
24. Watcom VX•REXX Project Folder	99
25. Watcom VX•REXX Development Environment: Window Layout	99
26. Watcom VX•REXX Development Environment: Event Code	100
27. DB2 Class Inheritance Diagram	102
28. DB2 Tables for Car Dealer Application	103
29. DB2 Table Definitions	104
30. DB2 Database Definition	104
31. DB2 Sample Table Load	105
32. Boxie the Cat	113
33. Using REXX to Update a DB2 BLOB	115
34. Using REXX to Fetch a DB2 BLOB	116
35. DB2 Definition for the Vehicle Table with Multimedia	117
36. Using Object REXX to Build and Store a DB2 BLOB	120
37. Vehicle Multimedia Window of Dr. Dialog GUI Application	122
38. DB2 Stored Procedure	134
39. DB2 Stored Procedure with Object REXX Shared Objects	134
40. DB2 Stored Procedure with Shared Objects: Server	136
41. DB2 Stored Procedure with Shared Objects: Gateway	137
42. DB2 Stored Procedure with Shared Objects: Client	138
43. Car Dealer Data Class Relationships	146
44. Directory Structure for Car Dealer Application	147

45.	DB2 Configuration Command File	148
46.	Car Dealer Application Configurations	149
47.	Car Dealer Application Overall Class Relationships	151
48.	Using the Local Directory	153
49.	Simple Car Dealer Installation Program	156
50.	Configuration File for FAT Persistence	158
51.	Configuration File for DB2 Persistence	158
52.	The Car Dealer Class	159
53.	IDL for the SOM Object Part	167
54.	IDL for the SOM Class PartMeta	168
55.	Object REXX PartBase Class for SOM	169
56.	Implementation Steps for SOM Object Part	171
57.	Car Dealer Show WPS Folder	174
58.	Car Dealer Customer View Folder	175
59.	Car Dealer Customer View Folder, Expanded	175
60.	Car Dealer Views	176
61.	Customer View Folder Populated by Drag and Drop	177
62.	Hacurs Home Page: Top Half	188
63.	Hacurs Home Page: Bottom Half	189
64.	Hacurs Home Page HTML Code	189
65.	Initial Design for the Car Dealer Application on the Web	191
66.	CGI Environment Variables (Extract)	193
67.	CGI Program to List All Parts	193
68.	Car Dealer Part List in Web Browser	194
69.	Object-Oriented CGI Program to List All Parts	195
70.	HTML Class for CGI Programs (Extract)	196
71.	Customer Search Form	197
72.	HTML for Customer Search Form	197
73.	Customer List in Web Browser	199
74.	Customer Details in Web Browser	200
75.	Tailored Web Server Administration File	202
76.	Car Dealer Start Program for the Web	204
77.	Car Dealer Common Interface Program	205
78.	New and Used Car List in Web Browser	206
79.	Vehicle Picture in Web Browser	207
80.	HTML Form for a New Customer and Car	209
81.	HTML Form for a New Work Order	211
82.	Final Design for Car Dealer Application on the Web	212
83.	Web Car Dealer Application Home Page	213
84.	Concurrency with Early Reply	218
85.	Concurrency with Message Objects	219
86.	Concurrency with Guard	220
87.	Example of Early Reply with Unguarded and Guarded Methods	221
88.	Sample Output of Early Reply with Unguarded and Guarded Methods	222
89.	Philosopher's Forks: Main Program	224
90.	Philosopher's Forks: Philosopher Class	225
91.	Philosopher's Forks: Fork Class	225
92.	Philosopher's Forks: Sample Output	226
93.	Philosopher's Forks: GUI Layout	227

94. Philosopher's Forks: GUI Run	228
95. Philosopher's Forks GUI: GUI Builder Logic	230
96. Philosopher's Forks GUI: Starter Class	231
97. Philosopher's Forks GUI: Philosopher Class	232
98. Philosopher's Forks GUI: Fork Class	233
99. Philosopher's Forks GUI: GUI Class for Dr. Dialog	234
100. Philosopher's Forks GUI: GUI Class for VisPro/REXX	235
101. Philosopher's Forks GUI: GUI Class for Watcom VX•REXX	237
102. Installation Program: User Interface	244
103. Installation Program: Progress Window	245
104. Installation Program: Config.Sys Update	246
105. Installation Program: Folder Creation	247
106. ObjectRexx Redbook Folder	247
107. Philosopher's Forks Folder	248
108. Installation Program: DB2 Table Definition	251
109. Installation Program: DB2 Table Load	252
110. Running the Sample Applications	253
111. Sample Customer Data (sampdata\customer.dat)	289
112. Sample Vehicle Data (sampdata\vehicle.dat)	290
113. Sample Work Order Data (sampdata\workord.dat)	290
114. Sample Service Item Data (sampdata\service.dat)	291
115. Sample Part Data (sampdata\part.dat)	291
116. Multimedia Data Definition File (media\media.dat)	292
117. Base Customer Class (base\carcust.cls)	293
118. Base Vehicle Class (base\carvehi.cls)	296
119. Base Work Order Class (base\carwork.cls)	297
120. Base Service Item Class (base\carserv.cls)	302
121. Base Part Class (base\part.ori)	304
122. Base Part Class as Subclass of a SOM Class (base\part.som)	307
123. Persistent Class (base\persist.cls)	308
124. Cardeal Class (base\cardeal.cls)	308
125. Configuration for File Storage (fat\carmodel.cfg)	310
126. File Customer Class (fat\carcust.cls)	311
127. File Vehicle Class (fat\carvehi.cls)	312
128. File Work Order Class (fat\carwork.cls)	314
129. File Service Item Class (fat\carserv.cls)	315
130. File Part Class (fat\carpart.cls)	316
131. Configuration for DB2 Storage (db2\carmodel.cfg)	317
132. DB2 Customer Class (db2\carcust.cls)	317
133. DB2 Vehicle Class (db2\carvehi.cls)	319
134. DB2 Work Order Class (db2\carwork.cls)	322
135. DB2 Service Item Class (db2\carserv.cls)	325
136. DB2 Part Class (db2\carpart.cls)	326
137. Command to Run the Car Dealer (\car-run.cmd)	327

Tables

1.	The Object REXX Collection Classes	33
2.	The Other Object REXX Classes	33
3.	Car Dealer Objects and Methods	40
4.	Methods Required by Every Data Class	44
5.	Methods Required for Customer Class	45
6.	Methods Required for Vehicle Class	45
7.	Methods Required for Part Class	46
8.	Methods Required for Service Class	46
9.	Methods Required for Work Order Class	46
10.	Relationships between the Car Dealer Objects	49
11.	Methods Required for AUI	62
12.	Methods Required for Menu	64
13.	Methods for Customer Persistent Storage in DB2	106
14.	Methods for Part Persistent Storage in DB2	107
15.	Methods for Service Item Persistent Storage in DB2	107
16.	Methods for Vehicle Persistent Storage in DB2	107
17.	Methods for Work Order Persistent Storage in DB2	107
18.	Icons of the ObjectRexx Redbook Folder	248
19.	Files of the CARDEAL Directory	254
20.	Files of the Sampdata Subdirectory	255
21.	Files of the Base Subdirectory	255
22.	Files of the FAT Subdirectory	255
23.	Files of the DB2 Subdirectory	256
24.	Files of the RAM Subdirectory	256
25.	Files of the AUI Subdirectory	256
26.	Files of the Media Subdirectory	257
27.	Files of the DrDialCD Subdirectory	257
28.	Files of the VisProCD Subdirectory	257
29.	Files of the VxRexxCD Subdirectory	257
30.	Files of the SOM Subdirectory	258
31.	Files of the StorProc Subdirectory	258
32.	Files of the WPS Subdirectory	258
33.	Files of the WWW Subdirectory	259
34.	Files of the Xamples Subdirectory	259
35.	Files of the Install Subdirectory	259
36.	Files of the PHILFORK Directory	260
37.	Subdirectories of the Car Dealer Application	288
38.	Subdirectories of the Philosopher's Forks Application	289

Special Notices

This publication is intended to help programmers use the new Object REXX language to create object-oriented applications. The information in this publication is not intended as the specification of any programming interfaces that are provided by Object REXX for OS/2 Warp. See the PUBLICATIONS section of the IBM Programming Announcement for OS/2 Warp for more information about publications that are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe on any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

The following document contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples contain the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

AIX	C Set ++
CUA	DB2
DB2/2	ES/9000
IBM	MVS/ESA
OS/2	OS/400
PowerPC	Presentation Manager
SOMobjects	System Object Model
ThinkPad	VisualAge
WebExplorer	Workplace Shell

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Windows is a trademark of Microsoft Corporation.

HockWare and VisPro/REXX are trademarks of HockWare, Inc.

Watcom and Watcom VX·REXX are trademarks of Watcom International Corporation.

Other trademarks are trademarks of their respective companies.

Preface

In this book we describe the new object-oriented language, Object REXX. We list all the incremental improvements that Object REXX offers over and above classic REXX and describe the object-oriented features included in Object REXX. To illustrate its capabilities, we develop some fairly large applications.

REXX has long had great strengths in the area of linking to other programs and services. Here we demonstrate Object REXX's ability to link to DB2/2 Version 2 to carry out sophisticated binary-large-object (BLOB) handling, as well as conventional record processing.

The number of OO applications alive and running is growing around the world. There is increasing demand to allow these objects to communicate with one another, even if they are written in different languages and run on different computers. The Common Object Request Broker Architecture (CORBA) standards specify a way in which this can be done. The number of implementations of CORBA by various vendors is burgeoning. IBM's version is the System Object Model (SOM). We show that it is easy for the Object REXX programmer to access and use SOM objects.

Object REXX also includes powerful facilities for concurrent programming. We show a graphical user interface (GUI) that exploits Object REXX's concurrent programming facilities.

Detailed syntax diagrams covering all of the new and changed features of Object REXX are included, with brief descriptions.

This book is intended for programmers who know and love REXX and would like to learn what the new facilities in Object REXX look like and the kinds of problems they can solve. It contains lots of sample code that we hope will provide a useful starting point for new projects. Programmers who currently use REXX to build large and complex systems will be well aware of its limitations in terms of splitting large programs into smaller, manageable components. Object REXX has excellent facilities that allow and encourage this process. We describe them and illustrate their use.

This book is also for programmers who would like to start learning and using OO techniques but do not have access to an OO language and compiler; or for those who *do* have access to one but find getting into it to be too complicated and alien. REXX is, above all, an accessible language. It is simple, obvious, and unintimidating; and Object REXX provides an easy entry into the world of objects.

How This Document Is Organized

The document is organized as follows:

❑ **Part 1, *Introducing Object REXX***

Part 1 is an overview of the OO facilities of Object REXX. It is also a description of why OO, in general—and Object REXX in particular—are such valuable and important technologies.

➤ **Chapter 1, *Introducing Object REXX***

In this chapter we introduce Object REXX and describe the importance of OO.

➤ **Chapter 2, *How Does Object REXX Implement OO?***

In this chapter we describe how Object REXX implements OO through objects, classes, and methods, including support for inheritance and polymorphism. It also touches on the Object REXX-provided class library.

❑ **Part 2, *The Car Dealer Scenario***

In Part 2 we illustrate a broad range of Object REXX's facilities by describing the way that a hypothetical software company might use them to design and implement a fairly realistic application for various car dealers.

➤ **Chapter 3, *The Car Dealer Application***

In this chapter we introduce a hypothetical software company, Hacurs. We describe the car dealer application that Hacurs wants to develop and the process that Hacurs goes through to design the system, using OO techniques. This chapter presents the Object REXX facilities that Hacurs decides to use in support of the implementation. Extracts of source code are included for illustrative purposes, while comprehensive source listings are included in Car Dealer Source Code on page 287.

➤ **Chapter 4, *ASCII User Interface***

In this chapter we describe how Hacurs designs and builds Object REXX classes and methods to implement a simple ASCII character-oriented user interface for the system. The company builds one class to manage the display of information on the user's screen and another to store, display, and interpret the many menus that the system requires. Anticipating the need for a future GUI interface, Hacurs uses OO design principles to isolate the application code from the user-interface code.

➤ **Chapter 5, *Persistent Objects on Disk***

In the base car dealer system, all updates to objects are lost when the application terminates. In this chapter, Hacurs designs and builds Object REXX classes and methods to add persistent storage behavior to the objects within the system. The object data is stored in flat ASCII files.

➤ **Chapter 6, *Graphical User Interfaces***

Chasing a new opportunity to sell its car dealer application, Hacurs builds and implements a GUI to it. The initial GUI package the company uses is Dr. Dialog; then VisPro/REXX and Watcom VX-REXX alternatives are added. The problems that arise when Object REXX class and method definitions are included in the code generated by these GUI builders are resolved.

➤ **Chapter 7, *Persistent Objects in DB2***

Seeing yet another opportunity to market the application, Hacurs develops new classes that give objects persistent storage in a DB2 database. The new methods can support large volumes of data by selectively loading only when needed and caching frequently used data in storage as objects.

➤ **Chapter 8, *Using Advanced DB2/2 Facilities***

Hacurs further extends the car dealer application by adding multimedia facilities. The code makes use of the powerful new BLOB handling facilities of DB2/2 Version 2 to store and retrieve the multimedia data. Audio, bitmaps, and video facilities are incorporated.

➤ **Chapter 9, *Data Security with Object REXX and DB2***

A serious concern that arises over the security of DB2/2 data accessed by dynamic SQL from client PCs is resolved by developing code that exploits DB2/2 Version 2's stored procedure mechanism.

➤ **Chapter 10, *Configuration Management with Object REXX***

A proliferation of different versions of the code required to meet different customers' needs threatens to get out of hand and result in a big code-maintenance burden. Hacurs develops a sophisticated system for managing many different code configurations within a multiple subdirectory structure, using different configuration files to pull the right pieces together. This allows common code to be reused without being cloned.

Hacurs develops a GUI Object REXX program that allows users to select the application configuration they need, and installs it.

➤ **Chapter 11, *Object REXX, SOM, and Workplace Shell***

Still another marketing opportunity arises, but to win the business, Hacurs needs to interface the car dealer Object REXX code to SOM objects. Hacurs develops a simple SOM object in C++ and modifies the Object REXX code to import and use this SOM object.

The OS/2 Workplace Shell (WPS) is SOM-enabled and can, thus, be accessed directly from Object REXX code by importing the WPS SOM classes. Hacurs experiments with this facility to build displays of car dealer objects in WPS folders on the desktop. This stimulates thinking about building objects in Object REXX that can be assembled with commodity objects for new applications.

➤ **Chapter 12, *Object REXX and the World Wide Web***

Hacurs decides to advertise its car dealer application on the World Wide Web, often called the *Internet*. It installs a Web server and creates a simplified version of the application to present car dealer data as Web pages. It uses the Common Gateway Interface (CGI) to invoke Object REXX programs from the Web server. The Object REXX programs dynamically create Web pages with the information from the database.

Any Web browser can point to the Hacurs server and interact with the car dealer application. An extension of the application even enables a Web browser user to add a car to the database and create a work order.

□ **Part 3, *Object REXX and Concurrency***

➤ **Chapter 13, *Object REXX and Concurrency***

In this part we describe the concurrent-processing facilities of Object REXX. After a short introduction, we solve with Object REXX the problem of the dining philosophers, a classic illustration of concurrent processing. The code to build a GUI application illustrating five philosophers sitting down to dine is developed and discussed. GUIs are developed in Dr. Dialog, VisPro/REXX, and Watcom VX·REXX.

□ **Part 4, *Installing Object REXX and the Sample Applications***

➤ **Chapter 14, *Installing Object REXX and the Sample Applications***

In this part we describe how to install Object REXX and both sample applications, the car dealer and the dining philosophers. Installation of the code and the setup for DB2 are explained in detail, including instructions on how to run the examples.

❑ **Part 5, *New Features and Syntax in Object REXX***

➤ **Chapter 15, *New Features in Object REXX and Migration***

This part contains a comprehensive set of syntax diagrams that show the new instructions, functions, classes, and methods that are a part of Object REXX, as well as the extensions made to REXX. The syntax diagrams are accompanied by explanatory text.

Differences between REXX and Object REXX are explained in a small migration section.

❑ **Appendixes**

The appendixes contain an extract of the source listings of the car dealer application and instructions on how to read the syntax diagrams.

Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this document.

❑ *Object REXX Reference for OS\2* [REXX.INF]

❑ *Object REXX Programming Guide for OS\2* [REXXPG.INF]

The two books listed above may not be available in hardcopy yet, but are shipped in online format with Object REXX.

❑ *The REXX Language, A Practical Approach to Programming*, by Mike Cowlshaw, published by Prentice Hall, 1990, ZB35-5100-01, ISBN 0-13-780651-5.

❑ *The Essential Client/Server Survival Guide*, by Robert Orfali, Dan Harkey, and Jeri Edwards, published by John Wiley & Sons, Inc., 1994, SR28-5572-00, ISBN 0-471-13119-9.

❑ *The Essential Distributed Objects Survival Guide*, by Robert Orfali, Dan Harkey, and Jeri Edwards, published by John Wiley & Sons, Inc., 1995, SR28-5898-00, ISBN 0-471-12993-3.

International Technical Support Organization Publications

- ❑ *OS/2 REXX: From Bark to Byte*, GG24-4199
- ❑ *Object-Oriented Databases, ObjectStore, Introduction and Sample Application*, GG24-4128

(This book is based on the same car dealer application that we use in our book.)

A complete list of International Technical Support Organization publications, known as *redbooks*, with a brief description of each, may be found in:

International Technical Support Organization Bibliography of Redbooks, GG24-3070.

To get a catalog of ITSO redbooks, VNET users may type:

```
TOOLS SENDTO WTSCPOK TOOLS REDBOOKS GET REDBOOKS CATALOG
```

A listing of all redbooks, sorted by category, may also be found on MKTTOOLS as ITSOPUB LISTALLX. This package is updated monthly.

How to Order ITSO Redbooks

IBM employees in the USA may order ITSO books and CD-ROMs using PUBORDER. Customers in the USA may order by calling 1-800-879-2755 or by faxing 1-800-445-9269. Most major credit cards are accepted. Outside the USA, customers should contact their local IBM office. Guidance may be obtained by sending a PROFS note to BOOKSHOP at DKIBMVM1 or by e-mail to bookshop@dk.ibm.com.

Customers may order hardcopy ITSO books individually or in customized sets, called GBOFs, which relate to specific functions of interest. IBM employees and customers may also order ITSO books in online format on CD-ROM collections, which contain redbooks on a variety of products.

ITSO Redbooks on the World Wide Web (WWW)

Internet users may find information about redbooks on the ITSO World Wide Web home page. To access the ITSO Web pages, point your Web browser (such as WebExplorer from the OS/2 3.0 Warp BonusPak) to the following:

```
http://www.redbooks.ibm.com/redbooks
```

IBM employees may access LIST3820s of redbooks as well. Point your Web browser to the IBM Redbooks home page:

```
http://w3.itso.ibm.com/redbooks/redbooks.html
```

ITSO Redbooks and Sample Code on the Internet

If you do not have World Wide Web access, you can obtain the list of all current redbooks through the Internet by anonymous FTP:

```
ftp ftp.almaden.ibm.com
cd /redbooks
get itsopub.txt
```

This FTP server also stores the sample code developed for this redbook. To retrieve the sample files, issue the following commands from the redbooks directory:

```
lcd d:\carinst <=== any local directory for installation files
binary
cd SG244586
mget *.*
ascii
get read.me
```

For IBM people without access to the external FTP server, the code is also available as OREXXRED PACKAGE on the OS2TOOLS conference disk.

To install the sample code, follow the directions in Chapter 14, "Installing Object REXX and the Sample Applications," on page 241.

About the Authors

Trevor Turton works in Johannesburg for IBM South Africa. He had the great idea of writing most of the book in the dialog style—Hanna, Curt, Steve, and the Hacurs company are his inventions. You can reach him at trevort@vnet.ibm.com.

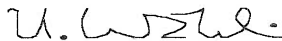
Ueli Wahli works at the IBM International Technical Support Organization in San Jose, California. For the past 12 years he has written IBM redbooks on all kinds of application development projects. His e-mail address is wahli@vnet.ibm.com.

Acknowledgments

This book would not have been possible without the help of the following people who contributed to the content of the book:

- ❑ Eddie Griborn from IBM Sweden ported the Philosopher's Forks application to VisPro/REXX and Watcom VX-REXX, and wrote about the concurrency and new features of Object REXX.
- ❑ Norio Furukawa from IBM Japan coded the implementation of persistence objects on disk and ported the car dealer GUI application to VisPro/REXX and Watcom VX-REXX.

Many thanks to Maggie Cutler for editing the book and making the dialog interesting. Thanks also to Jens Tiedemann, manager of the ITSO San Jose, for the investment of resources into the Object REXX projects on OS/2 and Windows NT and 95.



Ueli Wahli

Part 1



Introducing Object REXX

1

Introducing Object REXX

Object REXX is a new implementation of the system procedures language, REXX. Apart from numerous detailed improvements, this version of REXX includes a full set of OO facilities. It is now called *Object REXX*. This chapter outlines some of the good things that have been added to REXX with the OO version of the language.

Object REXX is available first in OS/2 for Intel processors, next in Windows NT and Windows 95, and possibly other operating systems and platforms in the future.

What's New in Object REXX?

Object REXX has many important new facilities; indeed, it is almost a new language. Chapter 15, *New Features in Object REXX and Migration*, on page 263 describes these in detail, and Chapter 2, *How Does Object REXX Implement OO?*, on page 17 contains an overview of how Object REXX has implemented its OO facilities. The concurrency capabilities of Object REXX are described in Chapter 13, *Object REXX and Concurrency*, on page 217.

Here are a few of the highlights. Object REXX has:

- ❑ A full set of OO facilities
- ❑ Direct access to SOM objects under OS/2
- ❑ Concurrency—the ability to do several things at once
- ❑ Improved ability to create subroutines with private variables
- ❑ Ability to embed source files using the *requires* command
- ❑ Ability to handle the error conditions that might arise in a called subroutine
- ❑ A *do over* command that visits every variable defined on a stem
- ❑ A *parse* command that has more case-handling options
- ❑ A *signal* command that can handle five new conditions

Object REXX has been designed to be upward-compatible with the previous versions of REXX. With a few minor exceptions described in *Migration Considerations* on page 286, all existing REXX programs should run under Object REXX with no change.

Despite its upward compatibility with prior versions, however, Object REXX is radically different from its predecessors. In classic REXX, every variable that the programmer created is conceptually a character string—even numbers appear to be stored this way. We say *conceptually* because, under the covers, REXX implementations are at liberty to store numbers in either integer or floating-point format, so long as they are always presented in string format when the programmer asks to see them. Since humans represent both numbers and text in string format when they communicate with one another, why shouldn't they continue to do so when talking to computers? This makes programming in classic REXX very simple and intuitive.

In Object REXX, every variable now refers to an object! String objects behave just as they have always done in classic REXX, and arithmetic can still be performed on strings that happen to contain numeric values; but Object REXX introduces a number of new object types and includes facilities for programmers to create even more of their own. We will be looking at these in some detail later. So while the internals have changed a great deal, Object REXX behaves very much the same as classic REXX used to—if you ask it to do the same things.

Perhaps we should mark the end of classic REXX's reign and the beginning of the reign of Object REXX with the proclamation that traditionally greets the death of a monarch and the automatic, immediate succession of his heir:

“The King is dead! Long live the King!”

Why REXX?

The REXX language is only about 15 years old but is already very widely used on IBM operating systems. The first version ran under VM/CMS only, but since then IBM has made REXX a standard component of the following operating systems:

- ☐ VM/ESA
- ☐ MVS/ESA
- ☐ OS/400
- ☐ OS/2
- ☐ PC DOS Version 7
- ☐ AIX (as a PRPQ)
- ☐ Netware (as a PRPQ)

Other vendors have developed REXX interpreters for various other operating systems. Also, many vendors have developed packages that are coded in REXX and/or generate REXX programs automatically. Some examples of these packages, each of which is a GUI builder and execution environment for REXX, are:

- ☐ VisPro/REXX
- ☐ Watcom VX-REXX
- ☐ Dr. Dialog

The thing that makes REXX so popular is that it is very easy to learn, and REXX code is easy to read and understand—compared to most other computer languages, that is! The language is *interpretive* in style, which means that there is no requirement to pass the source code through a compiler or linker before executing it. REXX programmers can change their code and test the changes immediately. The other great advantage to programmers is that REXX is nondeclarative. Programmers do not have to tell REXX how to store the variables they create. Conventional compiled languages such as COBOL and C do require declarations of this sort, and this roughly doubles the number of lines of code that have to be developed.

Even if we ignore the OO features that Object REXX contains, this new version of the language contains a number of significant improvements that make REXX easier to use and capable of producing more robust code.

Why Object Orientation?

Object orientation is the flavor of the year, perhaps of the decade. Most new language announcements that hit the press include the magic OO phrase, even if the applicability of OO to the product in question is sometimes unclear. Old languages such as C, COBOL, and Pascal have been extended to include OO features. Is OO a silver bullet that will solve all our programming problems, or is it just a fad?

The computer language that introduced OO concepts to the world is Smalltalk. This was originally designed in the 1970s as part of an experiment to see whether children could learn to use computers. We now know that the answer to that question is a resounding “yes!” (It is less clear, however, whether their parents can do likewise.) Smalltalk underwent significant change, but by 1980 it had the features that are indelibly associated with OO today:

- ❑ Objects grouped in classes
- ❑ Inheritance
- ❑ Polymorphism

These concepts are described in Chapter 2, *How Does Object REXX Implement OO?*, on page 17. But before we get into the nuts and bolts of how OO works, we should spend some time discussing the question of whether OO is worth doing at all.

The Productivity Problem

A clinical discussion of OO features does very little to explain *why* they are valuable. There is much talk today of the need for programmers who have been trained in conventional procedural languages such as COBOL to undergo a paradigm shift before they can start to understand and exploit the benefits that OO has to offer.

The benefits claimed for OO design and programming include much greater reuse of code, as well as simpler programs that are easier to understand and modify. Electronic computers have been around for about 50 years. Programmer productivity has improved radically over this time. Even so, the biggest inhibitor to the more extensive use of computers remains our inability to produce good, reliable code quickly enough to meet our users’ needs. The tools and techniques that we use today to develop computer applications are still very labor-intensive, when compared to those in other industries. Most people have heard the proud boast of the computer hardware industry:

“If the airline industry had been able to improve its technology as rapidly as has the computer hardware industry, today’s airliner would be able to fly anywhere in the world in half an hour and carry 10,000 passengers at a cost of \$1.”

Sounds impressive. Unfortunately, we in the computer software industry have not nearly as much to boast about. It has been said that:

“If the airline industry had improved its technology at the same rate as has the computer software industry, today’s airliner would be built from parts on the runway by the crew each time it flew, fueled with the finest Scotch whiskey, and used to haul garbage.”

Things are not really that bad in the software industry. Our technology has advanced rapidly and consistently since the advent of computers, at a rate that is impressive when measured against any criterion except one—our users’ needs. The biggest problem facing software developers is that computer hardware keeps getting cheaper and faster all the time. Applications that were technically possible but completely unaffordable 10 years ago are more than just affordable today; they are compulsory if a business is to compete in the current market.

Fortunately, there is a vast and rapidly growing number of off-the-shelf computer packages. Smaller businesses can often meet all their application needs from these packages. Larger businesses also make extensive use of packages but often need to supplement them with applications that support their core business. In many cases, a company’s core computer applications give it the competitive edge that enables it to grow and prosper.

The Reuse Solution

To summarize the previous section: There are not enough programmers and there is not enough time to handcraft all the code required to meet our users’ needs. The challenge is to deliver much more function, much more quickly. The only way out of the dilemma is not to try to develop all the code we need but to reuse existing code instead.

Programmers have been reusing code for a very long time. Early operating systems included subroutines to handle the complexities of driving I/O devices, and early languages such as Fortran (first built in 1957) included extensive libraries of subroutines that implemented the complex algorithms needed to calculate trig functions and logs. Most languages allow programmers to develop their own subroutine libraries to handle common requirements, and most information technology (IT) departments make use of these facilities (every installation has at least one date-handling subroutine, for example).

So if we already practice code reuse, what is so special about OO? Properly used, OO allows us to change the way we design and code applications, but to do so we must make a fundamental shift from the *procedural* to the *object-oriented* approach. Changing from procedural to OO application design can be difficult. The experiments in teaching children to use Smalltalk, referred to in *Why Object Orientation?* on page 6, showed that children can learn and use OO concepts quite eas-

ily, but for those of us who have been conditioned to design and code with procedural languages, the change to OO requires some unlearning.

Let us try to illustrate the differences between classic procedural design and OO design.

The Waterfall Method

Procedural design has converged on a process called the *waterfall method*. This consists of a series of steps. In theory, each step should be completed before the next is started. The steps are:

- ☐ Gather the business requirements
- ☐ Analyze the requirements
- ☐ Produce a high-level design
- ☐ Produce detailed specifications
- ☐ Code and unit test the specified modules
- ☐ System test the modules together

It has long been known that this approach has a serious drawback, inasmuch as the users have to express their needs fully and formally on paper and then wait 6 to 18 months before they get to see what the IT specialists thought they wanted. It is, in fact, very difficult for anyone to envisage an IT solution to a business need using just paper specs. Usually, the system has to be modified once the users understand how it works. However, the limitations of procedural languages strongly encourage this approach, and it is the norm.

The Spiral Method

Object-oriented tools can be used with the waterfall technique, but a more common approach is the *spiral method*. In this, IT specialists and the users plan to go through the design and implementation phases many times over before the project is complete. The analysts work with the users to identify the various business procedures they need to automate. They then work through the details of each procedure and document them in what is generally called a *use case*. Next, the coders build a small and simple prototype that implements the user interface with just enough logic behind that to make the interface behave as expected. There are no databases or even data models at this stage. The IT specialist and the users then work through the use case with the prototype. The users get an early idea of how the proposed system will help or hinder them in the execution of their responsibilities. They tend to become very involved and excited, then identify changes and new features that they need. In this way, it is also easier to see which features deserve a lower priority.

On the basis of this feedback, the designers revise their use cases and designs, and the coders modify the prototypes to implement the new behavior. The users work with the new prototypes and identify more changes. The entire process repeats several times, then the final version is fleshed out into a robust and reliable application, and delivered

to the users. Experience shows that applications designed in this way fit the users' real needs far better than is normally achieved with the waterfall approach.

Prototyping

Prototyping is not a new concept. The idea has been around for a long time. The problem has always been that the classic procedural languages such as COBOL, PL/I, C, and Assembler are not well suited to developing prototypes. It takes too long to build a prototype, and, once developed, the investment in the prototype code is so large that the programmers cannot afford to abandon it. It is very hard to make extensive design changes to procedural code, so the first prototype often ends up being the final product, regardless of how well it fits the users' needs. Further problems arise when several independently prototyped components must be integrated to form the complete application. They often do not fit together, and extensive changes may be required. It is exactly because of these problems that the waterfall method was developed. The users and analysts are required to anticipate every code module that will be needed and to ensure that all the components will fit together. Many experienced users view this as a shrewd maneuver on the part of the IT department, designed to shift the blame for humanity's inability to predict the future from the shoulders of the IT department to those of the users.

Object-oriented languages enable programmers to take a very different approach to building prototypes. Experience shows that OO prototypes are easier to modify and extend and can be changed to meet the users' changing perceptions of what they really need. While the parts of the overall application may be developed independently of each other, OO languages allow these different components to be integrated, forming a working whole with little disruption to any of the parts. The transition from prototype to production code is a smooth process, with few ugly surprises.

The Paradigm Shift

The fundamental difference between procedural and OO designs arises from the fact that procedural languages cannot be extended. Procedural language programmers can use only those features that were built into the procedural language by the vendor that supplies it. The programmer cannot add new commands or data types to the language, no matter how much these may be needed in a given situation.

Object-oriented languages, on the other hand, *are extensible*. Designers and programmers can add new data types to the OO language to meet their unique business needs. These are called *objects*. They can add new operations, called *methods*, to the OO language to manipulate existing or new data types. New objects can be built on top of existing data types within the OO language, and on top of other objects that the programmers have already defined.

If, for example, OO COBOL had been available 20 years ago, life would have been much simpler for the software vendors who introduced major new database management systems (DBMSs) at that time. They could have extended COBOL's capabilities to include support for their DBMSs by developing new class libraries. Lacking this ability, many chose to create completely new computer languages, called *4GLs*, to allow easy access to the features of their DBMSs. A new computer language is a major investment for the vendor that builds it, the programmers who learn to use it, and especially the companies that accumulate legacy code in it.

Procedural languages force the designer and programmer to follow a process known as *stepwise refinement*. The designer first specifies a business requirement at a high level. There are no features in the procedural language that can directly implement the objects described in this design or the actions that must be performed on them. The designer must break each object down into a collection of simpler objects and each action into a series of simpler steps. This process has to be repeated until the objects are so simple that they can be directly represented in the primitive data types supported by the procedural language and the actions can be equated to the primitive operations implemented by the procedural language. The entire stepwise refinement procedure takes place on paper, not in code. Only the final step in the process is captured as code and appears in the application. All prior steps in the process are captured on paper and are not delivered as part of the running application.

Suppose we compare the design and coding of an application in a procedural language to the growth of a tree. The high-level design would correspond to the trunk and major branches of the tree. The detailed designs would correspond to its smaller branches, spreading out into twigs. The actual code would correspond to its leaves. When the application is handed over to the maintenance programmers, they regard the code as the most important thing they get. The design documents usually do not correspond exactly to the code, because the users ask for changes late in the implementation process, and changing the design document is usually low on everyone's priority list. As time goes by and the application undergoes maintenance, the design documents are seldom updated. After a while they are so far out of step with the code that they are useless and are ignored.

To go back to our tree analogy, the maintenance programmers can now see only the leaves, not the branches or twigs that were used to join them all together. From the outside (the users' view) it still looks and behaves like a tree—for a while, but from the inside it becomes increasingly difficult to see how the whole thing hangs together.

Is this gradual loss of visibility and understanding of the program's structure important? Yes it is, vitally so. Every program has an invisible component that we can call the *flow of control*. It's the way the computer sees the program when it executes. This is the most important view of the program, because it determines absolutely what the

program does, regardless of what the programmers think it should do. Well-structured programs help the programmer to see what the flow of control will be and, hence, what the program will do. Most programs are not well structured—not so much because their contents are badly structured, but because of what they do *not* contain—the tree’s trunk, branches, and twigs, to return to our analogy. Maintenance changes start to have unexpected side effects. Someone saws off a branch without knowing what leaves it supports, someone bends a branch to support new leaves and inadvertently cuts off the flow of sap to some other leaves. After a while, the tree no longer looks anything like a tree from the inside—it looks like a bowl of spaghetti. Then it is time to throw the whole thing out and start again from scratch—and that is a waste of time and money.

Suppose we now compare the design and coding of an application in an OO language to the growth of a tree. Once again, the design would correspond to the trunk and branches of the tree. This time, however, the components of the design *can and should* be written into the code of the final application, rather than written on paper and then discarded. Because OO languages are extensible, the objects and actions described in the design can be written directly into code. If the design speaks of customers ordering, taking delivery, and paying for products, the programmers should create new object types called *Customer* and *Product*, as well as methods that allow product objects to be ordered, delivered, and paid for by customer objects. The high-level program logic can then reflect the high-level design because it talks about exactly the same objects and actions (methods) as does the design document.¹

In essence, we can turn the design and coding process on its head when we build OO applications. Instead of proceeding with stepwise refinement from a high-level design through successively lower-level, paper-based designs until we get down to the level of the procedural language and then writing the code, we can start by writing the design in code as if all the objects and actions it requires were already part of our target language. Then we use the language’s OO facilities to *define* what these objects are, and how they behave. As we define these objects and their behavior, we often find that there is still a gap between the level of abstraction at which we are working and the built-in features of our language. Once again, we boldly code our new

¹ People with experience of real-life application construction may at this stage be throwing up their hands in horror. Building new applications tends to generate a huge volume of paper, usually referred to as *The Documentation*. We do not suggest that a 1,000-page mound of paper be shoveled into the code. Most of the documentation exists to explain, criticize, measure, report, and mend the design. Entity-relationship diagrams, data-flow diagrams, action diagrams, and their ilk are a good way of representing design concepts graphically. Gantt charts are a good way of representing plans and progress graphically. All these things generate an amazing amount of paper (which is often pasted up on the walls to show the users how productive the designers have been), but they are not the design. It is our belief that a well-structured OO program that contains its own design will be no bigger than the equivalent program coded with no embedded design in a procedural language. The OO programmers will write much of their logic at a high level against smart objects, while procedural languages constrain us to write all our logic at a low level against dumb objects.

definitions in terms of lower-level objects and actions *as if they already existed*.¹ We will come back later and define these lower-level objects and actions and continue in this way until, at last, all the objects and actions we need are actually present in the OO language we are using.

Please note the use of the word *can* in the preceding paragraph. It is unfortunately quite possible for programmers trained in procedural languages to ignore the capabilities of OO languages to preserve design and to use OO in exactly the same way that they previously used COBOL or C to produce only the low-level code. Proper training and motivation are required if the transition to OO is to be fruitful.

Better Reuse from the OO Approach

The way we design and build applications using OO languages should, therefore, be very different from the way we build them using procedural languages. This change is the biggest one that procedural programmers and analysts must make when moving to OO. Why do we do it this way? What are the benefits?

- ❑ The biggest long-term benefit is that most of the application's design is encapsulated in its code. It cannot be discarded or ignored. All changes to the application automatically update the detailed design document because it is a living part of the code. This makes long-term maintenance of the application easier and more accurate.
- ❑ Much of the application logic is written in terms of high-level objects that correspond directly to the objects with which the users work. Programmers and users can speak the same language because they are speaking about the same objects and actions (methods).
- ❑ Less code is required because it deals with high-level smart objects, such as products, that can do complex things like get ordered by customers, rather than with dumb objects, such as integers, that can do only simple things like arithmetic.
- ❑ Objects like customers have their data and associated actions (methods) neatly packaged together in OO language definitions. It becomes much easier to locate and reuse customer objects and their associated behavior in other applications.

¹ We are not suggesting that we should embark on the design and implementation of a major system without careful analysis and planning. If we simply write a program as thoughts pop into our heads, the results will be as poor with OO languages as they are with procedural ones. We will get "stream of consciousness" programs, or what we might call *Kerouac code*. It may make for entertaining reading, but trying to make it work correctly will be much less fun. Good methods and tools are available to help the OO analyst identify the objects and methods that should form the basis of a new system. However, in this section we are trying to identify what is *different* about OO analysis and design, not what is the same.

- ❑ When programmers reuse an object, they do not need to know how it works internally. The author of the object can carry out maintenance on it to add new data or functions without impacting any of the programs in which this object is used.
- ❑ OO languages all come with an extensive library of built-in classes, which can be used to define new objects. These inherit a wealth of high-level function. Much of the tedious low-level coding required to build an application can be eliminated by making use of these class libraries.

Communities of Cooperative Objects

In dealing with the benefits of OO, we have so far restricted ourselves to those that are currently being enjoyed and reported by installations that have made the switch. However, there is a sea of change taking place right now in how objects will be exploited in the future, and it is going to affect all of us.

Bloated PC Software

It is a well-known fact that “shrink-wrap” applications are getting bigger and better every year—with most of the emphasis falling on *bigger*. Ten years ago, the most sophisticated spreadsheet package came on a single floppy. Now it takes a diskette caddy to load the simplest package. Have our needs changed that much over the past 10 years? Has life really become 10 times more complex? Why is shrink-wrap software so bloated? It costs the vendors a fortune to build applications of this size and complexity, so you can be sure they are not doing it for fun. Also, a fact of life with software is: the bigger, the buggier.

Ten years ago, PC enthusiasts sneered at the “big, clumsy, slow” programs that ran on mainframes and rejoiced in their tiny, nippy applications. They have stopped talking about it. Many are watching what is happening in numb silence.

PC software is suffering from “creeping featuritis.” One vendor puts in a great new feature, all the competitors put it in, too, as well as a few more unique features of their own. More bullets on the side of the shrink-wrap box. More check-boxes in the endless assessments that PC software magazines run. More entries in the already crowded menu bar. More chapters in the phone-book-sized product manual. More days on the education program. More space on the hard drive. More RAM tied up. More bucks on the bill. Where is it all going? Is this trip really necessary? Most of us use only a fraction of the features of the PC software we run.

Standard Software Components

There is another way, and it is based on the notion of building software from a host of standard, reusable components. We touch on this in *The Productivity Problem* on page 6, with those not-so-funny comparisons between the IT and airline industries. When hardware engineers want to build a system, they pick standard parts out of a catalog and wire them together. Little or none of the componentry that they need has to be invented on the fly. The parts are highly standardized and uniform in their behavior, and there are few surprises when they are clicked together. Generally, the new system works.

Software engineering is light-years away from this model. The way we handcraft code today is reminiscent of the way our ancestors used to manufacture¹ products before the industrial revolution. Programming is still in the “cottage industry” phase of development.

Liberating Objects from Applications

All this is due to change soon—indeed, is changing already. Objects have shown that they can deliver specific functions while encapsulating all their internal workings so that the programmers using them do not have to know what goes on inside. Currently, the object’s horizon is limited to the application that contains it. If you want to build a lot of function into an OO application, you have to put a lot of objects into it. About six years ago, the folk in the emerging world of objects realized that objects would become much more useful if objects could be used across different applications, even if the applications were written in different languages—and even if they ran on different computers, maybe even under different operating systems (this democratic vision is not shared by all in the industry).

The CORBA Standard

To make any of this happen, standards are an absolute necessity. A cross-industry standards group called the *Object Management Group* (OMG) was formed in 1989 to develop and publish standards in this area. The OMG has been very industrious and successful, and its membership has risen to over 500. Almost every company involved in building objects is in the OMG and is busy enabling its object software to conform to the OMG standards. The biggest “umbrella” standard from OMG is called the *Common Object Request Broker Architecture* (CORBA). As with all standards, the longer the name, the more arguments and reconciliations went into its formation. The CORBA standard was widely and hotly debated by the members of the OMG, and what came of the crucible is case-hardened steel.

¹ *Manufacture*: verb, from Latin *manus*, “a hand,” and *facto*, “I make.”

IBM's CORBA-compliant object broker implementation is called the *System Object Model* (SOM). It is a standard component of OS/2 and AIX and will soon be standard in MVS/ESA and OS/400, too. Probably every popular operating system will have a CORBA-compliant object broker from one vendor or another by the end of 1996.

We pick up on this theme again in Chapter 11, *Object REXX, SOM, and Workplace Shell*, on page 161, and in more detail in *Applications Assembled from Components* on page 179. The topic is far too big to fit into the confines of this book. Several excellent publications already exist on this topic alone. We particularly recommend *The Essential Distributed Objects Survival Guide* by Robert Orfali, Dan Harkey, and Jeri Edwards (see full reference in *Related Publications* on page xxvii).

So Why Object REXX?

In the preceding sections we have reviewed how successful REXX has been and how useful OO facilities are. The marriage of the two is an obvious and welcome step, bringing to the programmer a language with the strengths of both. Object REXX is likely to be widely and enthusiastically embraced by the REXX programming community for the following reasons:

- ❑ It's free! Everyone is talking about OO nowadays, but getting access to an OO language costs money. Object REXX is distributed as a standard component of OS/2 at no extra charge, and Object REXX is a full-function OO language. What better way to get your feet wet in the OO puddle than by using the language you know and love?
- ❑ Object REXX lets you learn about OO incrementally. While it enables you to build totally nonprocedural code, you can also start adding OO features to existing procedural programs. You do not have to abandon your legacy REXX code or your existing skill base.
- ❑ The standard REXX trace and debug facilities are still available, even for OO code. You can step through your programs line by line, displaying and setting variables as you go. This makes it easy to understand what is going on.
- ❑ Object REXX includes new features that make it much easier to build structured and modular applications. REXX is being used to build some very large and complex systems, and these new structuring capabilities are most welcome.
- ❑ One of the key benefits that OO gives is reuse. REXX programmers are, in general, very familiar with the reuse approach. Programs coded in other languages can be invoked directly from within REXX programs. Commands for other programming environments, such as XEDIT under VM and the EPM editor, DB2/2

and CPI-C under OS/2, can be embedded in REXX code. Most REXX programmers are comfortable with a “mix-and-match” approach. Object REXX extends the range of resources available to the REXX programmer to include objects developed in Object REXX itself, as well as OS/2 SOM objects.

- It is easier to interact with the OS/2 WPS from Object REXX. Professional application installation routines do more than just copy files into a new subdirectory; they also instruct WPS to build the folders and other icons the users need to drive their systems.

Learning how to exploit OO does not have to be a white-knuckle experience. Object REXX provides an easy path into the world of objects, building on and enhancing existing REXX skills.

2

How Does Object REXX Implement OO?

Object REXX has a very comprehensive set of OO facilities, including multiple inheritance and meta classes (see *Methods* on page 28). It has support both for static class and method construction through embedded declaratives, and for dynamic class and method construction through messages that may be issued at runtime to the built-in Class and Method classes. Here we use only the static, declarative forms.

Object REXX can import and use SOM objects and classes (see Chapter 11, *Object REXX, SOM, and Workplace Shell*, on page 161). The Object REXX manuals referenced in *Related Publications* on page xxvii contain an excellent description of OO concepts and how Object REXX implements them. We give only a brief and incomplete outline of these capabilities here for the reader's convenience.

We need to start by emphasizing that the magic of OO does not lie in its definition. Many people have labored over descriptions of OO, seeking the philosopher's stone that will transform dull gray code into glistening gold in the concepts of objects, classes, inheritance, and

polymorphism. Try as you will, you will not find it there. What is important about OO is the changes it *allows*—but does not *require*—designers and programmers to make in the way they structure programs. We have tried to explain this in *Why Object Orientation?* on page 6.

Objects

All of us deal with objects every day of our lives. Things like faucets, toasters, refrigerators, cars, telephones, photocopiers, fax machines, and televisions are objects. We use objects to do things. We give commands to objects. We might open a faucet, push down the cook lever on a toaster, start a car, depress an accelerator or a brake pedal, turn a steering wheel, dial a number on a telephone or fax machine, or press a channel change or mute button on a TV remote control.

Objects must be able to obey the commands we issue. They need some built-in, predefined behavior. In the OO world these are called *methods*.

Object REXX uses the ~ (tilde) operator to invoke a method on an object.

Invoking methods on an object

```
car~start
car~turn('right')
car~speed(55)
```

The word preceding the tilde is the object, and the word following it is the method. Those familiar with classic REXX can think of invoking a method as something similar to invoking a function. Consider the following code:

Invoking methods compared to functions

```
aString = 'Hello, World'
say aString                (gives: Hello, World)
say reverse(aString)       (gives: dlrow ,olleH)
say aString~reverse        (gives: dlrow ,olleH)
```

If every object we encountered differed from all others and had its own unique set of commands, we could never cope with the daily demands of living. Humans have learned to standardize the way similar objects behave and are controlled. Different car models made by different manufacturers in different countries all have similar controls and respond in a similar way when these controls are used. Even if we have to fly to a distant country, we can still operate the cars we find there with reasonable success. Every car is different. Each has a unique number plate and engine and chassis serial numbers. Each

has its own unique collection of scratches and bumps and little quirks, such as the way it hesitates when you floor the accelerator at 50. But cars, and trucks for that matter, behave similarly enough that drivers can move from one to another and cope.

Classes

The world in which programmers must operate is also populated by objects. These objects, too, have their own unique attributes and built-in behavior. Programmers cannot cope with the diversity of objects that they must manage unless they simplify and standardize the behavior and appearance of these objects as much as possible. Quite often, programmers will impose a greater degree of standardization than exists in the real world. A program may insist, for example, that every human has a surname and one or more given names. While this is a common practice in some European countries, in Nordic countries it is not, and the people on other continents have very different practices. We have learned to live with generalizations like these so that programmers need cope only with a subset of the problems the real world contains.

In order to cope with the innate complexity of the world, programmers must seek and impose similarities in behavior across groups of related objects. In OO terminology, a group of related objects is called a *class* or *type*. Once they have identified a class of objects, programmers can define and code the routines, or *methods*, that give these objects their common behavior.

Object REXX uses *directives*, placed at the end of the program, to define classes and methods. A directive starts with two colons (:).

Directives for a class definition

```
::class Car
::method start
...
::method turn
...
::method speed
...
```

Note: Class definitions can also be placed into separate files using the *::requires* directive. (See *The Requires Directive* on page 76.)

Inheritance

So far, we have done little more than coin some trendy new OO terms to describe well-established programming practice. Now we start to add something new and exciting. It's called *inheritance*. It stems from the fact that, although we want to group objects into classes and enforce a common behavior across all of them, some stubbornly refuse to fit a common mold. Cars and dump trucks have similar controls to drive them, but dump trucks have extra controls to manage the dumping mechanism. How can we cope with this irritating diversity? We might be tempted to build the code needed to manage cars, then clone it and extend it to handle trucks. It gets the job done, and we score extra brownie points if our productivity is measured in lines of code, but it creates an extra maintenance burden that will last for as long as the code runs.

Object-oriented languages offer an elegant way of coping with the problem of similar but different classes. Given the problem described above, we could define a *Car* class that implements the behavior common to both cars and dump trucks (for example, starting, steering, and stopping), then define a new class called *DumpTruck* that is a subclass of the *Car* class (see Figure 1).

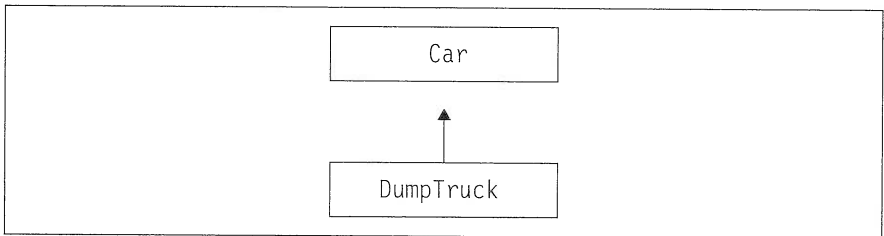


Figure 1. *Car and DumpTruck Class Inheritance Diagram*

The Object REXX class directives might look like the following:

Class directives for inheritance

```
::class Car
::method start
...
::class DumpTruck subclass Car
::method dump
...
```

A subclass *inherits* all the behavior (attributes and methods) of its parent class but can add new attributes and methods of its own. We can add to our *DumpTruck* class just the new behavior that is unique to dump trucks—the ability to dump. So any dump truck objects that we create will automatically inherit all the methods they need to be

driven and will also have the methods they need to dump. We have achieved the equivalent of cloning code without actually cloning code. Maintenance is simplified.

A nice side effect of this approach is that when we add new methods to the base *Car* class to handle new behavior such as fuel consumption, all of its subclasses automatically inherit these new methods as well.

Abstract Classes

But what would happen if we needed to add some new methods to the *Car* class that we did *not* want its subclasses to inherit? Suppose we needed to add information about a car's trunk capacities and optional extras—sidewalls, two-tone color schemes, and such? We could abstract from the *Car* and *DumpTruck* classes all the attributes and methods we want them to have in common and put them in a new *abstract* class called *Vehicle*. We would make both *Car* and *DumpTruck* subclasses of the *Vehicle* class (see Figure 2).

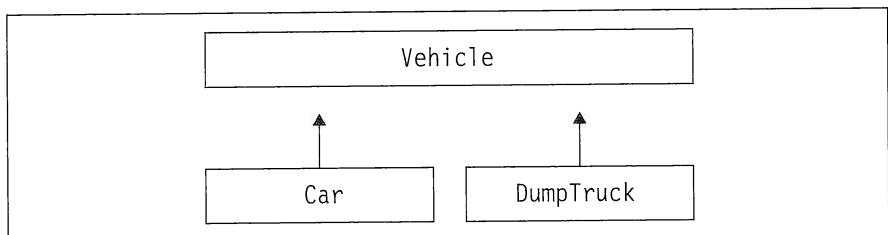


Figure 2. *Abstract Class Inheritance Diagram*

Each would inherit all the common behavior it needs from the base *Vehicle* class, and we could then add to each the behavior that it alone requires. We might never create an object directly from the *Vehicle* class. It would serve just as a handy place to keep common behavior. Since we do not change the names of the *Car* and *DumpTruck* classes, none of the code that deals with them will be affected. All the changes we make are hidden inside the class definitions. This is an example of *encapsulation*, one of the major benefits of OO.

The Object REXX directives required in this case might look like this:

Class directives for an abstract class

```
::class Vehicle
::method start
...

::class Car subclass Vehicle
::method trunk_capacity
...

::class DumpTruck subclass Vehicle
::method dump
...
```

Can we take this further? Suppose the need arises to deal with trucks other than just dump trucks. How would we handle this situation? In Figure 3, we abstract the behavior that is common to dump trucks and tanker trucks and put it in a new abstract class called *Truck*. We then define *TankerTruck* and *DumpTruck* as subclasses of *Truck*. They both inherit the behavior of the base *Vehicle* abstract class and the behavior of the *Truck* abstract class, then each adds its own unique behavior to its own class.

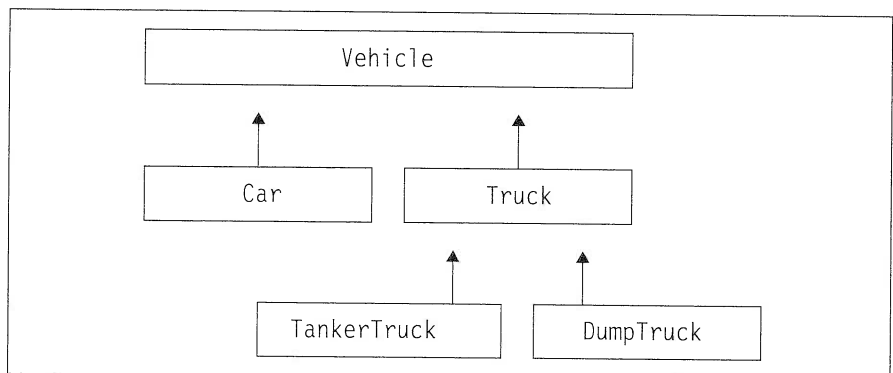


Figure 3. *Multilevel Class Inheritance Diagram*

The Object REXX directives required in this case might look like this:

Class directives for multilevel inheritance

```
::class Vehicle
::method start
...
::class Car subclass Vehicle
::method trunk_capacity
...
::class Truck subclass Vehicle
::method hitch_horse
...
::class DumpTruck subclass Truck
::method dump
...
::class TankerTruck subclass Truck
::method fill_tank
...
```

We can continue in this way to create as many levels of inheritance as we need.

Multiple Inheritance

The inheritance story could have ended here, but Object REXX takes it further. The Smalltalk language allows each class to have only one parent class from which it can inherit behavior. Object REXX allows classes to inherit from one or many parent classes. Only one can be the direct parent. The other parents are called *mixin* classes. Like abstract classes, they are not used to generate instances. They serve only as containers for attributes and methods that other classes can inherit from them.

Suppose we need to add information about engines to our vehicle fleet. In the old class structure, engine information was contained in the *Vehicle* class. We observe that the same sort of engine is often used in different types (classes) of trucks, and some engines are common between light trucks and cars. We want to separate out the engine information from the rest of the vehicle, which we will call the *Body*. We might do this as shown in Figure 4.

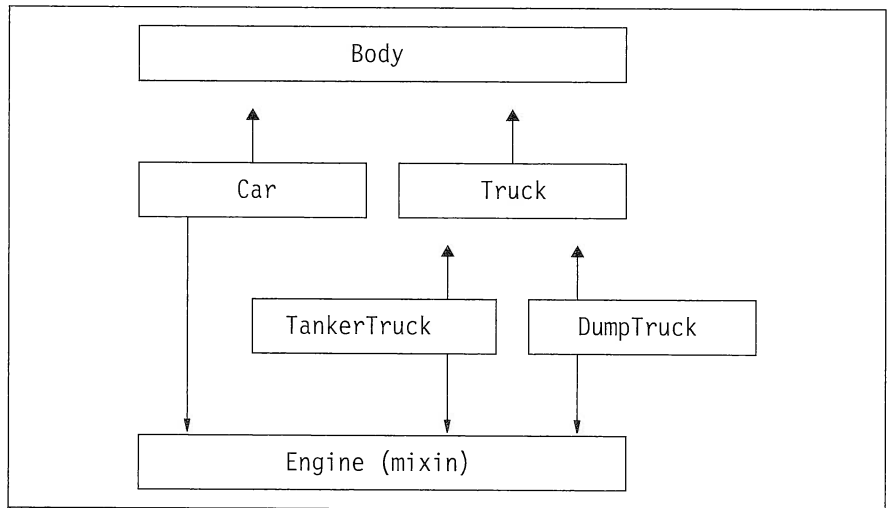


Figure 4. *Mixin Class Multiple Inheritance Diagram*

The Object REXX directives required in this case might look like this:

Class directives for multiple inheritance

```

::class Body
::method rattle
...
::class Engine mixinclass Object
::method start
...

::class Car subclass Body inherit Engine
::method trunk_capacity
...
::class Truck subclass Body
::method hitch_horse
...

::class DumpTruck subclass Truck inherit Engine
::method dump
...

::class TankerTruck subclass Truck inherit Engine
::method fill_tank
...

```


The old *Vehicle* abstract class has disappeared, its attributes and methods split into two new abstract classes called *Body* and *Engine*. *Body* becomes the direct parent of *Car* and *Truck*, while *Engine* becomes a mixin class. Each vehicle now obtains its body and engine behavior from two different classes.

Object REXX Variable Pools

In classic REXX, by default each .cmd file has its own variable pool. Variables set by code within the .cmd file are available to all other code within the same .cmd file. The programmer can change this by coding the procedure instruction after a label, for example:

```
aProcedure: procedure
```

When *aProcedure* is called, REXX creates a new and private variable pool for *aProcedure*. This remains in effect until *aProcedure* terminates. In this example, none of the code within *aProcedure* can access any of the variables set by the code in the command file, and vice versa.

The programmer can obtain a limited degree of exposure of the variable pool external to *aProcedure* by using the *expose* option. For example:

```
aProcedure: procedure expose variable1 variable2 stem.
```

The variables and stems listed after the *theexpose* keyword map directly onto the corresponding variables in the variable pool that was active when *aProcedure* was called. Changes that *aProcedure* makes to these exposed variables remain in effect when *aProcedure* terminates. Existing REXX programs work the same way in Object REXX as they do in classic REXX, to preserve compatibility.

Objects are new in Object REXX, and they are handled differently. Each object usually has several variables, or *attributes*, associated with it. If you have 100 employee objects active, each one may have its own name, number, address, and other attributes. Object REXX associates a separate variable pool with each object.

An object's attributes can be accessed only by the methods that are defined within the object's class; in OO terminology, all data is private or *encapsulated*. Each method must specify which of the object's attributes it needs to access by listing them on an *expose* instruction immediately after the method directive (*::method*).

Example of a method

```
::method tag
  expose name address salutation
  separator = 'at:'
  return salutation name separator address
```

In this example, the variables *name*, *address*, and *salutation* are part of the associated object's variable pool. All the other methods in the class that contains this method may access and set these variables if they first expose them. Any variables a method uses that are not in its expose list are local variables and are discarded as soon as the method terminates. In this example, *separator* is a local variable.

Note: If an object inherits methods from different classes, it will have different variable pools in each class. A method defined in one class cannot share a variable with a method in another class. If methods need to share information, the owner of the variable must implement methods to get and set this variable, and the would-be sharer must invoke them.

The benefit that flows from this arrangement is that different groups can build and maintain different classes quite independently. Multiple inheritance can make bedfellows of complete strangers. If a new class claims parentage from two independently developed classes, there is no danger that the accidental use of the same variable name in the two parent classes will cause collisions and corruption of the variable. The methods of each parent class will continue to operate in its separate variable pool. This approach mirrors the way in which SOM classes manage their variables.

The down side of this arrangement is that it is a little difficult to split the definition of what the programmer may regard as a single class over more than one source file. Each class definition must be completely contained in a single file. Different files will, therefore, contain different class definitions. The methods in these files may be pooled by using inheritance, but they will not be able to gain access to one another's variables except through get and set methods created specifically for this purpose.

As in SOM, Object REXX provides a very simple way of creating get and set methods for a given variable. With the code:

```
::method aVariable attribute
```

Object REXX will automatically create both a *get* and a *set* method for *aVariable*. One can then get and set the value of *aVariable* by coding:

```
something = anObject~aVariable
```

```
anObject~aVariable=(aValue)
```

```
anObject~aVariable=aValue
```

Note: Unless a *SIGNAL ON NOVALUE* or a *SIGNAL ON ANY* instruction is included in the code, the methods may happily appear to use variables to which they actually have no access, if these variables happen to lie in a separate variable pool, for reasons described above. It is probably a good discipline to include the *SIGNAL ON NOVALUE* instruction in code while it is being debugged, and to leave it in while it is being used in production.

Object Instances

We have introduced objects and classes, but how do we actually create and delete objects within a class? Individual objects are often called *instances*.

Object Creation

Most OO languages provide a *new* method (operator) to create an instance of a class. This is also how object creation is implemented in Object REXX.

Object creation

```
mycar = .Car~new
...
::class Car
::method start
...
```

Note: The *Car* class defined using the class directive (*::class*) is available in the program as *.Car*, and the *new* method is invoked against this class object. In Object REXX, even classes are themselves objects.

We will often need to initialize the variables of a newly created object. Object REXX automatically invokes the *init* method of a new object, if an *init* method has been defined. The *init* method can accept parameters to initialize object variables and set additional variables to default values.

Initializing a new object

```
mycar = .Car~new(12345, 'Ford', 'Mustang')
...
::class Car
::method init
  expose serialNumber make model saleDate
  use arg serialNumber, make, model      /* parameters of new */
  saleDate = date('s')                  /* initialize variable */
::method start
...
```

Note: The new *USE ARG* statement is used to assign values to variables from the arguments. This is more effective than parsing the arguments and works for any objects passed in as parameters in the method call. See *USE (New)* on page 275 for more details.

Object Destruction

In Object REXX there is no explicit way to delete an object. Object REXX supports automatic garbage collection—that is, objects without any references (variables pointing to them) are removed from memory periodically under system control.

The program can remove references to objects by assigning another value to a variable or by dropping the variable:

```
mycar = .Car~new
...
drop mycar      /* object is subject to garbage collection */
```

Methods

Methods of a class are defined in the directives section of the program immediately after the class directive. We will often want methods to return a result that can be used by the invoking program, but this is not compulsory.

Methods can be invoked in two ways, through a single tilde (~) or through a double tilde (~~). When a double tilde is used, any result returned by the method is disregarded, and the object to which the method was applied is returned instead. This allows several methods to be applied to a single object in one statement, in a procedure known as *chaining*.

Chaining of method operations

```
car = .Car~new(...)
car~~start~~speed(55)~~for(5m)~mileage
...
::class Car
...
```

Note: Since the double tilde does not return a result, the subsequent operations work on the same car object until the *mileage* method returns the miles driven in 5 minutes.

Private and Public Methods

Methods invoked from the main program or from other classes are specified as *public* methods. They define the interface of the class, that is, all the possible operations this class can perform.

Methods used only within the class—that is, they are invoked only from other methods of the class—are *private* methods.

By default, methods are public; the *private* keyword is used to define a private method.

Public and private methods

```

::class Car
::method milage                /* public method for users */
    self~calculate             /* - invoke private method */

::method calculate private    /* private method */
    expose time speed          /* - not available to users */
    return time * speed / 3600 /* - used by other methods */

```

Classes and Instance Methods

So far, we have spoken about methods operating on objects. While this is generally the case, some methods cannot operate on specific objects because, for example, the method's purpose may be to *create* a new object, and the code calling the method cannot point it to this new object because it does not exist until after the method has run. When we deal with “normal” methods that operate on objects to do things like print them, shred them, or delete them, we speak of *instance* methods. They operate on objects, which are also known as *instances* of their class. When we deal with methods that we cannot pass a specific object to, we call them *class* methods.

This may sound rather technical, but when it comes to writing the code, the distinction will usually be very obvious. Making a method do something to an existing object requires an instance method; otherwise it must be a class method.

Instance methods usually handle the data of an individual object, whereas class methods handle data about the whole class, such as counting the number of objects in the class or managing a collection of all the objects.

Meta Classes

We have spoken about classes inheriting methods from their parents and from mixin classes. Although we did not mention it at the time, a subclass inherits both the instance and the class methods of its direct (and mixin, if any) parents. We spoke of abstract and mixin classes as a handy way to store behavior that can be inherited by a new subclass. Now we introduce meta classes. Like abstract and mixin classes, they are a handy place to store methods and attributes for other classes to inherit. The wrinkle is, when a new class inherits from a meta class, the meta class's *instance* methods become the inheriting class's *class* methods—along with any other class methods it inherits from its direct parent.

If this sounds complex, it is! But seldom will an Object REXX application programmer need to use meta classes. Direct inheritance usually gets the job done, with mixins less often required. The people who really need meta classes are the programmers who build OO languages like Object REXX. They could have kept meta classes hidden and used them for their own purposes only, but they chose to share them with the world. There are good reasons for doing this. If the feature is there, why not make it available? People have built some very complicated and sophisticated systems using OO languages in the past, and we believe that Object REXX will be no exception. Also, Object REXX needs to interface with the rest of the world. Probably the most important OO interface that Object REXX has is its linkage into SOM, as described in Chapter 11, *Object REXX, SOM, and Workplace Shell*, on page 161. SOM is the equivalent of a telephone exchange for objects, allowing them to find and talk to each other, no matter where they may be. SOM requires its programmers to deal explicitly with meta classes when class methods must be defined. It is probably a good idea for the Object REXX community to understand what meta classes are all about, and to be able to use them when required.

Polymorphism

Polymorphism is the rather cumbersome name given to a very simple idea that almost every computer language offers. It is the notion that a single operator symbol, like +, −, *, or /, can be used against operands of different types, such as short integer, long integer, short float, or long float. The language compiler (or interpreter) determines the type of operand that is involved and uses one of the many available machine instructions to carry out the appropriate operation. So whenever two numbers are added together, a plus sign is written between them, regardless of their data type.

Object-oriented languages enable programmers to define their own functions and operators (methods) for the new data types (classes) they create. For example, a *draw* method can be defined for the Shape class and, therefore, for all its subclasses (triangle, rectangle, circle, etc.). The draw method can then be invoked against an object of every subclass of the Shape class. Object REXX invokes the implementation of the draw method according to the class of each object.

A very nice example of polymorphism may be found in `complex.cmd` in the Object REXX sample subdirectory and the associated `usecomp.cmd` that invokes it. This code creates a class of complex numbers and defines operators to carry out simple arithmetic on them. The programmer is free to choose any method name to denote the addition of complex numbers—*complexAdd*, for example, which would require the following syntax:

```
a = b~complexAdd(c)
```

Instead, he or she wisely chooses the plus operator (+) for this purpose. This allows the programmer using the complex class to code:

```
a = b + c
```

This is, of course, a very familiar notation, and programmers will find it easy to apply these new methods to the new domain of complex numbers, even though complex numbers are not a part of standard REXX.

Just to show that Object REXX permits very useful things without much code, we show below the Object REXX method for adding complex numbers. By way of introduction for those who did not major in math, each complex number has two parts, called *real* and *imaginary*. Each of these two parts is a perfectly normal number. Combined, we can use them to do things like position a point on a graph, where we need to know how far to the right it is and how high. These two independent properties can be stored separately in the real and imaginary parts of a single complex number.

The Object REXX method for adding complex numbers

```

1  ::class complex public
2  ::method '+'
3      expose real imaginary
4      use arg adder
5      if arg(1,'o') then
6          return self
7      tempreal = real + adder~real
8      tempimaginary = imaginary + adder~imaginary
9      return self~class~new(tempreal, tempimaginary)
```

- 1 We use the `::class` directive to define the class of complex numbers.
- 2 We use the `::method` directive to start the definition of the + method.

- 3 We give each complex number two attributes, called *real* and *imaginary*. We *expose* them so the method can use them.
- 4 The `+` method normally works on two complex numbers. The first is the object in front of the `+` and the second is the object after it. We access the first object through the built-in name *self*. We access the second through the *use arg* statement and use the local variable name *adder* to reference to it.
- 5 If *arg(1)* is omitted, we do not have to do any addition.
- 6 We just return the object to which this prefix (`+`) applies (*self*).
- 7 We get the real part of *adder* and add it to the real part of the complex number we are dealing with.
- 8 We get the imaginary part of *adder* and add it to the imaginary part of the complex number.
- 9 We make a new complex number to return to the caller. We need to find the class of the object we are dealing with, because the method that makes new complex numbers is a *class* method (see *Classes and Instance Methods* on page 29). *Self~class* gets the class of the object, and *class~new* invokes the *new* method of the complex class.

The class library that is supplied with Object REXX (see *The Object REXX Class Library*, below) makes extensive use of polymorphism. For example, the method name `[]` may be used to refer to an element of any type of collection, be it from the Array, Bag, Directory, List, Queue, Relation, Set, Stem, or Table class. The `[]` notation has been long and widely used in languages like C to denote subscripting of arrays, so the Object REXX convention exploits and reinforces an association that many programmers already have.

Programmers are encouraged to follow the same convention when they create new methods. If it does something analogous to an existing method of another class, give it the same name. It is easier to remember the name when needed, and it is easier to guess what the method does when only the name is known.

The Object REXX Class Library

Every OO language worthy of the name comes with a set of class definitions. These do a wealth of useful things and spare the OO programmer reinventing the wheel. Object REXX is no exception. Many of its classes relate to managing collections of data. These are called the *Collection* classes and are shown in Table 1. Another group provides a variety of useful functions to the programmer, listed in Table 2. These are very terse lists; for details, please see the Object REXX manuals referenced in *Related Publications* on page xxvii.

Table 1. The Object REXX Collection Classes

Class Name	Purpose
Array	A sequenced collection
Bag	A nonunique collection of objects, subclass of Relation
Directory	A collection indexed by unique character strings
List	A sequenced collection that supports inserts at any position
Queue	A sequenced collection that can accept new items at its start or end
Relation	A collection with nonunique objects for indexes
Set	A unique collection of objects, subclass of Table
Table	A collection with unique objects for indexes

Table 2. The Other Object REXX Classes

Class Name	Purpose
Alarm	Generates asynchronous messages at specific times
Class	A technical class to create new classes
Message	Supports the deferred or asynchronous sending of messages
Method	A technical class to dynamically create new methods
Monitor	Manages the forwarding of messages
Object	A technical class to manage all objects
Stem	A collection indexed by unique character strings
Stream	Supports input and output operations
String	Supports operations on character strings
Supplier	Supplies the elements of a collection one by one

As with any OO language, the Object REXX class library is an extremely valuable asset and will richly repay careful study. We can never claim to know an OO language until we have a fair idea of what its class library contains.

The Object REXX Class Library Browser

Most OO languages contain a facility called a *class browser*, a program that presents the class library to the programmer on request. It supports lookup by class name or by method name. Object REXX provides this facility through its online reference facility, which should be installed when Object REXX is installed. With its hypertext links, it provides far more information than does the usual class browser. Of course, it can display only the built-in Object REXX class library. There is at this stage no equivalent facility for browsing programmer-defined classes.

We implemented two simple experimental class browsers in Object REXX. They are available as `browsc1s.cmd` and `browser.cmd` in the `Xamples` subdirectory of the car dealer application.

Part 2



The Car Dealer Scenario

3

The Car Dealer Application

In this chapter we introduce the Hacurs software company and pick up the story of how it uses Object REXX to implement a car dealer application. We look at the objects required for this application and find out how the classes built into Object REXX (its class library) can be used to help construct them.

Introducing the Hacurs Company

It is all too easy to make a book about a computer language read like a catalog of washing machine parts. As we go through the features of Object REXX, we are going to try to bring them to life by showing how useful they are to a fictional, but not unrealistic, small software company called *Hacurs*. This company was started one year ago by three friends—Hanna, Curt, and Steve. They studied computer application design and programming together at college, and after graduation they all joined the same company and worked in its IT department. They often spoke of starting their own little software company, and after two years of corporate life, they agreed to do it. They decided to design and develop applications for OS/2. They had used C and C++

for some of their college assignments, but most of their corporate experience was based on coding REXX. They recognized that REXX is an extremely powerful and easy-to-use language and chose it as their preferred development language.

Their company name is derived from their own names but also stands for their main line business—Handy Applications Coded Using REXX. Hacurs signed up with the IBM Developer Assistance Program (DAP) and the Developer Connection for OS/2 (DEVCON). This gives them access to lots of useful information, as well as some very useful development tools.

The Car Dealer Opportunity

“Hey, team,” yelled Curt as he banged in through the door of the Hacurs office late one afternoon, “we’ve got our breakthrough! I spent most of today with Trusty Trucks looking at their requirements for a car dealer system. They are really keen to automate this part of their business, and I’ve pretty near convinced them that we can build a system that will meet their needs, and that we can do it fast.”

“That’s wonderful,” said Hanna.

“Great going!” exclaimed Steve.

“What do they want?” asked Hanna.

“They service vehicles—cars and trucks,” Curt answered as he put down his bag and sat at his desk. “I tried my hand at developing a use case with them to describe their business process. I captured it on my ThinkPad.”

Curt pulled his ThinkPad from his bag, plugged it in and powered it on. Once it had booted up, he opened a view of his project subdirectory and dragged an icon to the EPM editor.

“This is what we came up with,” he said [see Figure 5 on page 39]. “Now, before you start criticizing, remember this is the first use case that I’ve built. We wrote up the steps that have to take place, and then I identified the nouns by making them bold, and all the verbs by making them underscored.

1. **Trusty Trucks** draws up a list of the **parts** it has in stock.
2. **Trusty Trucks** also defines the **services** it offers and lists the **parts** each service needs.
3. **Customers** bring in their **vehicles** for servicing.
4. **Trusty Trucks** records the **customer** and **vehicle** details on a **work order** and itemizes the **services** required.
5. **Service staff** carries out the specified **services** on the **vehicles**.
6. **Clerical staff** prepares **bills** based on the **work orders**.
7. The **customers** pay their **bills** and claim their **vehicles**.

Figure 5. *Car Dealer Application Use Case*

“We decided not to mark every noun,” said Curt. “Some just didn’t seem useful to us. All the nouns we highlighted are candidates for objects in the application design. And all the verbs we highlighted are candidates for methods.”

“Well that looks very simple and straightforward to me, Curt,” said Hanna, “although I’m sure it will turn out to be a lot more complicated when we get down to the details.”

“Should we try to draw up a list of the objects you have identified and their related methods?” asked Steve.

“OK,” said Curt. He copied the text of his use case and edited out all words except the highlighted ones. “This brings up a question,” he noted. “If I can remember back to my high-school grammars, most sentences have a subject, a verb, and an object. Both the subject and the object are nouns. But when we come to attach methods to objects, does the verb get associated with the subject or the object of the sentence? For example, in the first item I’ve got

‘**Trusty Trucks** *draws up* a list of the **parts** it has in stock.’

“Should *draws up* be the method of ‘Trusty Trucks’ or of ‘parts’?”

“Of ‘parts’, I think,” answered Steve. “The Trusty Trucks object uses the method, but the parts object must implement it. It deals with parts data.”

“OK, let’s use that approach and see what happens,” said Curt.

The Hacurs team worked together on this task. After some thought, they derived the table presented in Table 3.

Table 3. Car Dealer Objects and Methods

Object	Method
A list of parts	Draw up
Services	Define
Parts	List for each service
Vehicles	Bring in
Customer	Record the details
Vehicle	Record the details
Services	Itemize on a work order
Vehicles	Get services
Bills	Prepare
Bills	Pay
Vehicles	Claim

“It looks like we’ve got some nouns left over,” said Hanna. “Trusty Trucks, the Stores department, Service staff, and Clerical staff acted as subjects but never as objects in the use case sentences.”

“That’s interesting,” said Curt. “I discussed these with Trusty Trucks. We recognized that we could identify objects corresponding to various divisions within the company and store them in the database. Trusty Trucks couldn’t see any point in doing so. I suggested that we could capture these in a field within each transaction, to act as an audit trail in case they ever needed to know who did what. They could see the potential value of doing that, but they plan to have a paper audit trail of each transaction and decided against keeping it in the database.”

“We’re starting to see the value of the use case discipline,” said Steve. “It makes you take into account those loose bits and pieces that might otherwise be overlooked in the design. Even if you eventually decide to ignore them, it’s good that you had to think about them.”

“Good point, Steve,” said Hanna. “And Curt, I think you’ve done a great job collecting this information and getting to understand what Trusty Trucks needs. Is this all we have to do? Do we create a class for each of the objects we’ve defined, and a method for each of the verbs?”

“I’m afraid there’s a lot more to it than that,” responded Steve. “We have to decide on the shape that we want our application to take. There’s a lot of technical issues that we still need to discuss.”

“Like what?” asked Hanna.

“Like what kind of user interface we must develop,” Steve answered, “and what database manager we should use.”

“Or if we use a database manager at all,” added Curt.

The Application Model

Hanna, Curt, and Steve sat around a table together, going through the requirements for the car dealer application and trying to identify the objects they would choose to implement in Object REXX. After a couple of hours of work, they came up with five objects that seemed to play a dominant role:

- ☐ Customer
- ☐ Vehicle
- ☐ Part
- ☐ Service
- ☐ Work order

“This is it,” said Curt. “Customers bring their vehicles in for various services. Trusty Trucks records the services each vehicle needs in a work order. Each service requires a standard amount of labor and parts. Those are the objects we have to model. The relationships between the objects look like this.” Curt drew a sketch on the whiteboard [see Figure 6].

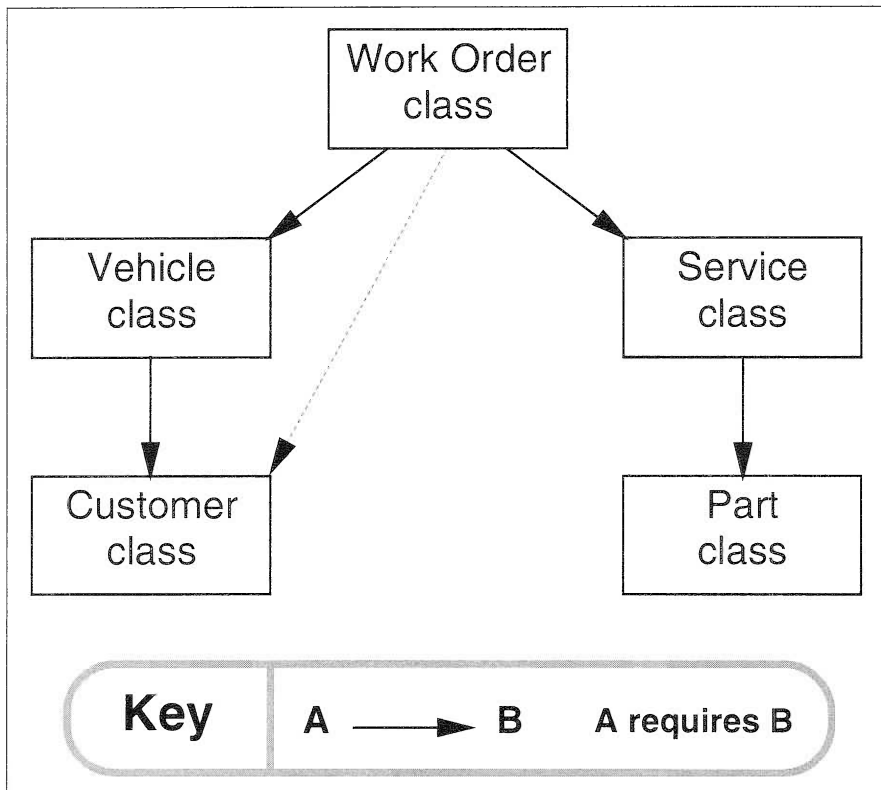


Figure 6. *Car Dealer Data Class Relationships*

“You don’t need a line from the *Work Order* class to the *Customer* class,” said Steve. “Each work order points to a vehicle, and each vehicle has an owner, and that’s who the customer is.”

“Not necessarily,” said Curt. “Suppose someone rents a truck and bends a fender. He or she might decide to take the truck in to get it fixed, rather than return it, dented, to the rental company. The customer is the renter, but the owner of the vehicle is the rental company.”

“That sounds pretty unlikely to me,” said Steve. “Anyhow, Trusty Trucks wouldn’t know that it’s a rented truck. They would capture the name of the person who brought the truck to them as the owner. All they care about is who’s going to pay them.”

Hanna broke in with a suggestion: “We can sort out this detail later. Make the line from the *Work Order* class to the *Customer* class dotted, and let’s carry on.”

“How about labor—don’t we need that as an object?” asked Steve.

“I don’t think so,” said Hanna. “The only thing we know about it is the standard labor charge for each service. We wouldn’t have any attributes to store in labor if we made it an object.”

“But there are different types of labor, and they charge out at different rates,” said Steve.

“Maybe so, but Trusty Trucks doesn’t want to record that kind of detail in its service records,” said Curt. “Let’s not make this more complicated than it has to be. We have to get a solution working fast if we want to get the business.”

“OK, let’s take those objects as our first cut,” said Hanna. “What attributes do we need to store for each one?”

“I’ve kept a list of the attributes as we went along, and they look like this,” said Curt, laying out a sheet of paper [see Figure 7].

01 customer		01 service	
05 custnum	smallint	05 itemnum	smallint
05 custname	char(20)	05 labor	smallint
05 custaddr	char(20)	05 description	char(20)
		05 servpart	occurs 20 times
01 vehicle		10 partnum	smallint
05 serialnum	integer	10 quantity	smallint
05 custnum	smallint		
05 make	char(12)	01 workorder	
05 model	char(10)	05 ordernum	smallint
05 year	smallint	05 custnum	smallint
		05 serialnum	integer
01 part		05 cost	integer
05 partnum	smallint	05 orderdate	char(8)
05 price	smallint	05 status	smallint
05 stock	smallint	05 workserv	occurs 20 times
05 description	char(15)	10 itemnum	smallint

Figure 7. *Car Dealer Object Attributes*

“That looks like a mixture of COBOL and SQL,” said Steve.

“Never mind, it gets the job done,” said Curt.

“You’ve got repeating groups in service and work order,” Steve noted. “We’ll have to normalize¹ the data.”

“Not necessarily,” said Hanna. “The collection classes in Object REXX allow an object to have attributes that are arrays, lists, sets, bags, directories...”

“OK, OK—you’ve made the point,” said Steve. “But when we come to store persistent objects in a relational database, we’ll have to normalize the data.”

“Also not necessary,” chimed in Curt. “The new binary-large-object (BLOB) support in DB2 Version 2 would allow us to store the repeating group as an array in a single BLOB column.” Seeing Steve’s look of concern, he added, “I would also feel more comfortable if the database was normalized, but the objects in storage don’t have to look exactly the same.”

Methods and Variables

“OK, if those are the objects we have to model, what comes next?” asked Hanna.

“We need to work out which methods each object must support and the variables they need,” said Curt. “I also kept a note of those as we went through the use cases. First, every object type...”

“You mean ‘class,’” interrupted Steve.

“Uh—yes, class,” agreed Curt. “OK, each class that manages the objects we’ve identified [see Figure 6 on page 41] needs the basic CRUD methods: create, read, update, and delete. Then whenever there’s a relationship between two different objects, we need a method to maintain it. Who owns which vehicle, for example. We need to be able to track changes in ownership without having to delete the old vehicle and capture it all over again under a new customer.”

“Why?” asked Steve. “You said we should keep it simple. Vehicles don’t change ownership that often. Why not discard the old vehicle record and capture a new one?”

“What if there’s a query relating to work done on the vehicle before it changed hands?” asked Hanna. “If we delete the old vehicle record we would lose any references we had to it in the work order history data.”

¹ *Normalization* of data is a term used in database design. In simple words it makes individual tables of a database nonredundant and all columns of a table nonrepeating and dependent on the key only.

Steve nodded, so Curt carried on: “Some of these methods relate to a specific object—update and delete, for example. These would have to be implemented as *instance* methods. But others don’t relate to a specific object. We would have to implement those as *class* methods.”

“Can you give us an example?” asked Hanna.

“Sure,” said Curt. “When we create a new object, we can’t send the create message to the object because it doesn’t yet exist. So we have to send the message to the object’s class instead, and it returns the new object to us. And when we want to search our customer set by name, we can’t send the message to a particular customer object, we have to send it to the class instead. The class method would come back with a customer object—or a list of customer objects if more than one has the search name, or maybe an empty list if the search fails.”

“Speaking of searching customers, how can we find all the customer objects that exist within the customer class?” asked Steve.

“We haven’t found any built-in way of doing that,” replied Hanna. “We could maintain a variable for the *Customer* class that consists of the set of all customer objects. Suppose we call it *extent*. Whenever a new object is created, Object REXX automatically calls the object’s *init* method. This is normally used to initialize the new object’s instance variables. It could invoke a class method that puts the new object into the set of all objects created. And likewise for the other objects that we need to keep track of.”

“That’s smart,” said Curt. “And we can have another class method that removes the reference to the object from the class’s extent attribute when the object is deleted.”

“Right,” said Hanna. “Let’s get to work and draw up tables of all of the methods we’ll need for each class. We won’t worry about how we store the objects on disk in this version. Let’s just concentrate on managing the objects in storage.” [See Tables 4–9.]

Method	Type	Purpose
initialize	Class	Initialize the extent variable
extent	Class	Return an array of all objects of the class
add	Class	Add a new object to the extent
remove	Class	Remove an object from the extent
init	Instance	Initialize a new object
setnil	Instance	Clear out the object record
delete	Instance	Delete the object from the class
detail	Instance	Return object details, formatted
makestring	Instance	Default ID for this object
display	Instance	Display object data on standard output

Table 5. Methods Required for Customer Class

Method	Type	Purpose
number	Instance	Return the number of the customer
findNumber	Class	Find a customer given the number
findName	Class	Return an array of customers matching name
heading	Class	Return a heading for output
name	Instance	Get or set the customer's name
address	Instance	Get or set the customer's address
update	Instance	Update the customer's data
addVehicle	Instance	Add a new vehicle to the customer
removeVehicle	Instance	Remove a vehicle from the customer
checkVehicle	Instance	Does this vehicle belong to the customer?
getVehicles	Instance	Return the customer's vehicles
findVehicle	Instance	Return a specific vehicle of the customer
addOrder	Instance	Add a work order to the customer
removeOrder	Instance	Remove a work order from the customer
getOrders	Instance	Return all work orders for this customer
ListCustomer-Short	Class	List customers on standard output
ListCustomerLong	Class	List customers with their vehicles

Table 6. Methods Required for Vehicle Class

Method	Type	Purpose
serial	Instance	Return the serial number of the vehicle
make	Instance	Get or set the vehicle's make
model	Instance	Get or set the vehicle's model
year	Instance	Get or set the vehicle's year
update	Instance	Update the vehicle's attributes
makemodel	Instance	Return make and model formatted
getOwner	Instance	Return the owner of the vehicle
setOwner	Instance	Set the owner of the vehicle
deleteOwner	Instance	Set the owner of the vehicle to nil

Table 7. Methods Required for Part Class		
Method	Type	Purpose
findNumber	Class	Return the part's number
heading	Class	Return a heading for output
number	Instance	Return the serial number of the part
price	Instance	Return the price of the part
description	Instance	Return the description of the part
stock	Instance	Return the stock level of the part
increaseStock	Instance	Increase the stock level of the part
decreaseStock	Instance	Decrease the stock level of the part
ListPart	Class	List all parts on standard output

Table 8. Methods Required for Service Class		
Method	Type	Purpose
findNumber	Class	Return the service's number
heading	Class	Return a heading for output
number	Instance	Return the number of the service
laborcost	Instance	Return the labor cost of the service
description	Instance	Return the description of the service
usesPart	Instance	Tell service it uses this part
getParts	Instance	Return the parts used by this service
getQuantity	Instance	Return the quantity used of this part
getPartsCost	Instance	Sum cost times quantity of parts used
getWorkOrders	Instance	Return work orders with this service
ListService	Class	List all services on standard output

Table 9. (Part 1 of 2) Methods Required for Work Order Class		
Method	Type	Purpose
findNumber	Class	Return the work order's number
newNumber	Class	Issue a new work order number
findStatus	Class	Return work orders of given status
number	Instance	Return the number of the work order
cost	Instance	Return the cost of the work order
date	Instance	Return the date of the work order
setstatus	Instance	Set the status of the work order

Table 9. (Part 2 of 2) Methods Required for Work Order Class

Method	Type	Purpose
getstatus	Instance	Get the status of the work order
getstatustext	Instance	Get the status of the work order as text
getCustomer	Instance	Get the customer of the work order
getVehicle	Instance	Get the vehicle of the work order
addServiceItem	Instance	Add a service item to the work order
removeService-Item	Instance	Remove a service item from the work order
getServices	Instance	Return services of this work order
getTotalCost	Instance	Compute the total cost of the work order
checkAndDecreaseStock	Instance	Issue the parts required for the services
generateBill	Instance	Return array of output lines of bill
detailcust	Instance	Return customer and vehicle details
makeline	Instance	Return work order details, formatted
ListWorkOrder	Class	List the work orders on standard output

“Wow! That’s a long list of methods,” said Curt, looking at the tables they had produced. “Aren’t we making this much more complicated than it needs to be?”

“I don’t think so,” answered Hanna. “The books on object orientation warn that you need a lot of methods to get the job done, but they say that the methods must be very short. Some say that if a method is longer than 30 lines, it’s too long. I read that in connection with Small-talk. I guess it’s too soon to say if the same limit should apply to Object REXX.”

“What’s the benefit of having lots of silly little methods, each of which does very little?” asked Curt. “Why not lump functions together to make fewer, bigger methods?”

“It’s like making bricks rather than prefabricating walls,” said Steve. “The simpler each method is, the more likely you’ll be able to reuse it for other purposes. And the more complex it is, the less likely you’ll be able to use it again.”

“Hmm,” mused Curt. “It sounds good, but I’ll reserve judgment on that until we’ve built our first application and I can see how it works out in practice.”

Relationships Among Objects

“I know it’s getting late, but I’d like to spend a little time talking about the relationships that we need to implement between the different objects,” said Hanna. “This is the way I see it.” She quickly constructed a list and showed it to the others. It read:

- ☐ A customer can own one or more vehicles
- ☐ A vehicle can be involved in many different work orders over time
- ☐ A customer can be involved in many different work orders
- ☐ Each work order requires one or more services
- ☐ Each service requires zero or more parts

Hanna asked, “Which of these relationships do we have to keep track of? And from which end?”

“What do you mean, ‘from which end?’” asked Steve.

“If we’re given a customer, do we need to know which vehicles he owns?” asked Hanna.

“Yes!” chorused Curt and Steve.

“And if we’re given a vehicle, do we need to know to which customer it belongs?”

“Yes!” chorused Curt and Steve again.

“Then maybe we need to put a list of vehicles owned into each customer object, and an owner attribute into each vehicle,” said Hanna.

“Wait a minute,” said Curt, “that doesn’t sound possible. If the vehicle object contains the customer object, which in turn contains the vehicle object, which one will really contain the other? Will we put the system into a perpetual loop trying to do what we tell it?”

“No,” smiled Hanna. “Objects never actually contain each other, they just contain references to each other. The objects themselves are all kept in Object REXX’s system storage. When you assign an object to a variable, you’re actually just storing a *pointer* to the object in the variable.”

Steve chimed in too: “And if you call a subroutine or method passing a huge BLOB as an argument, the system passes just a pointer to the BLOB.”

“Cute,” said Curt. “So how do we actually store the relationships that you spoke about? I seem to recall that there is a List class built into Object REXX.”

The Object REXX Collection Classes

“There’s a whole lot of collection classes built into Object REXX, including Array, Bag, Directory, List, Queue, Relation, Set, and Table,” said Steve. “All of them can be used to store sets of related information. All of them have several methods in common, and all have their own unique capabilities. We’re spoiled for choice—it’s almost embarrassing!”

“OK—so which should we use?” asked Hanna.

Realizing that this would take some time, the Hacurs team phoned out for pizza and went in detail through each of the relationships that Hanna had identified.

“So this is what we’ve agreed,” said Curt, wiping some tomato sauce from the handwritten table and presenting it to his teammates for their approval [see Table 10].

Table 10. Relationships between the Car Dealer Objects				
First Object	Second Object	From 1st to 2nd	From 2nd to 1st	Type
Customer	Vehicle	Set	Attribute	1:m
Customer	Work order	Set	Attribute	1:m
Vehicle	Work order	'none'	Attribute	1:m
Work order	Service	Relation	Relation	m:m
Service	Part	Set	'none'	m:m

“That’s cryptic!” exclaimed Steve. “What does it all mean?”

“It’s simple, really,” replied Curt. “The first two columns list different types of object. The third column shows how we record the relationship from the object in the first column to the object in the second. If we don’t, the entry is ‘none’. Otherwise it’s the name of the Object REXX class we agreed to use. The object that carries the relationship is stored as an attribute in the first object. The fourth column is the same as the third, except the other way around. It shows how we store the relationship in the second object back to the first. In most cases I’ve written just *attribute*. This means that there’s only one object of type one associated with the second object, so we don’t have to store a list, only a single object pointer. The fifth column shows the type of relationship we model. We distinguish between many-to-many (m:m) and one-to-many (1:m) relationships.”

“Why don’t we record the services that use a certain part?” asked Hanna. “Trusty Trucks is not interested in that information, so there is no need to carry it,” replied Curt, “and we handle the work orders from the customer directly, without going through the vehicle,” he added.

“Now that we see it all,” said Steve, “do we really have to use the relation class to implement the relationship between work orders and services? Wouldn’t it be simpler to use the same collection class for all our relationships?”

“Maybe, but this will look better on our CVs¹,” replied Curt with a smile.

“I’m more worried about our paychecks than our CVs!” muttered Hanna.

Example of relationship between customer and vehicle classes

```

::class Customer
...
::method init                                /***** NEW CUSTOMER *****/
  expose customerNumber cars                /* each customer object */
  use arg customerNumber                    /* has a customer number */
  cars = .set~new                           /* and a set of cars */
...
::method addVehicle                          /***** ADD NEW VEHICLE ****/
  expose cars                               /* new cars are added to */
  use arg newcar                            /* the set of cars */
  cars~put(newcar)
...

::class Vehicle
...
::method init                                /***** NEW VEHICLE *****/
  expose serialNumber owner                 /* each vehicle points to */
  use arg serialNumber, owner              /* the owner (customer) */
  owner~addVehicle(self)                   /* and adds itself there */
...

```

Object Creation and Destruction

“Let’s talk through the life-cycle of these objects and make sure they can all be created when needed and discarded when their work is done,” suggested Hanna.

“I think we’ve covered that,” said Curt. “We listed the methods that are common to every object [see Table 4 on page 44] and these include *init* to create new objects and *delete* to throw old ones away. We also plan to define an *extent* set as a class variable in each class to keep track of all the objects we have defined within that class. And we plan to have an *add* and a *remove* method for each class. *Add* will save a pointer to each new object in the extent when it’s created, and *remove* will drop it when it’s discarded. The *init* and *delete* instance methods will invoke the *add* and *remove* class methods.”

¹ CV, curriculum vitae, or resumé, a short account of one’s career and qualifications.

“That’s fine for keeping track of objects in storage,” responded Hanna, “but what happens when the user powers-off the PC? Are all the objects lost?”

“Ah! Now you’re talking about object persistence,” said Curt. “That’s a big topic, and this isn’t the right time to start getting into it. We’ve covered a lot of ground today, and I for one am getting tired.”

Hanna glanced at her watch. “You’re right, it is getting late. OK guys, let’s call it a day. Thanks for giving up your time for this project. It will be a big one if we manage to close the business. And with any luck we’ll be able to sell the same solution to a number of different companies. This could turn out to be the milk-cow application we need to keep our paychecks rolling in. Sweet dreams!”

Maintaining the set of objects of a class

```
.Customer~initialize                /* prepare the class */
cust1 = .Customer~new(101,'Steve') /* create some customers */
cust2 = .Customer~new(102,'Hanna')
.Customer~ListCustomerShort        /* list all customers */

::class Customer

::method initialize class           /****** class methods *****/
  expose extent                    /* prepare the set of cust. */
  extent = .set~new                /* in variable "extent" */
::method add class                  /* add new customers to set */
  expose extent
  use arg aCust                    /* Arg passed from new/init */
  extent~put(aCust)                /* - add it to the set */
::method ListCustomerShort class   /* list of all customers */
  expose extent                    /* iterate over extent */
  do aCust over extent             /* - call instance method */
    aCust~display                  /* for each customer */
  end

::method init                       /****** instance methods *****/
  expose customerNumber name       /* initialize variables */
  use arg customerNumber, name     /* - from arguments */
  .Customer~add(self)              /* add itself to the extent */
::method display
  expose customerNumber name       /* display cust. variables */
  say 'Customer: number='customerNumber 'name='name
```

Note: This simple example shows how instance methods can invoke class methods (*init* invokes *add*), and class methods can invoke instance methods (*ListCustomerShort* invokes *display*). The separation is very logical; operations at the class level are implemented as class methods using the *class* keyword in the method directive, and operations at the individual object level are instance methods.

Implementation of the Model in Memory

Figure 8 shows the object model with class and instance variables for the sample car dealer application.

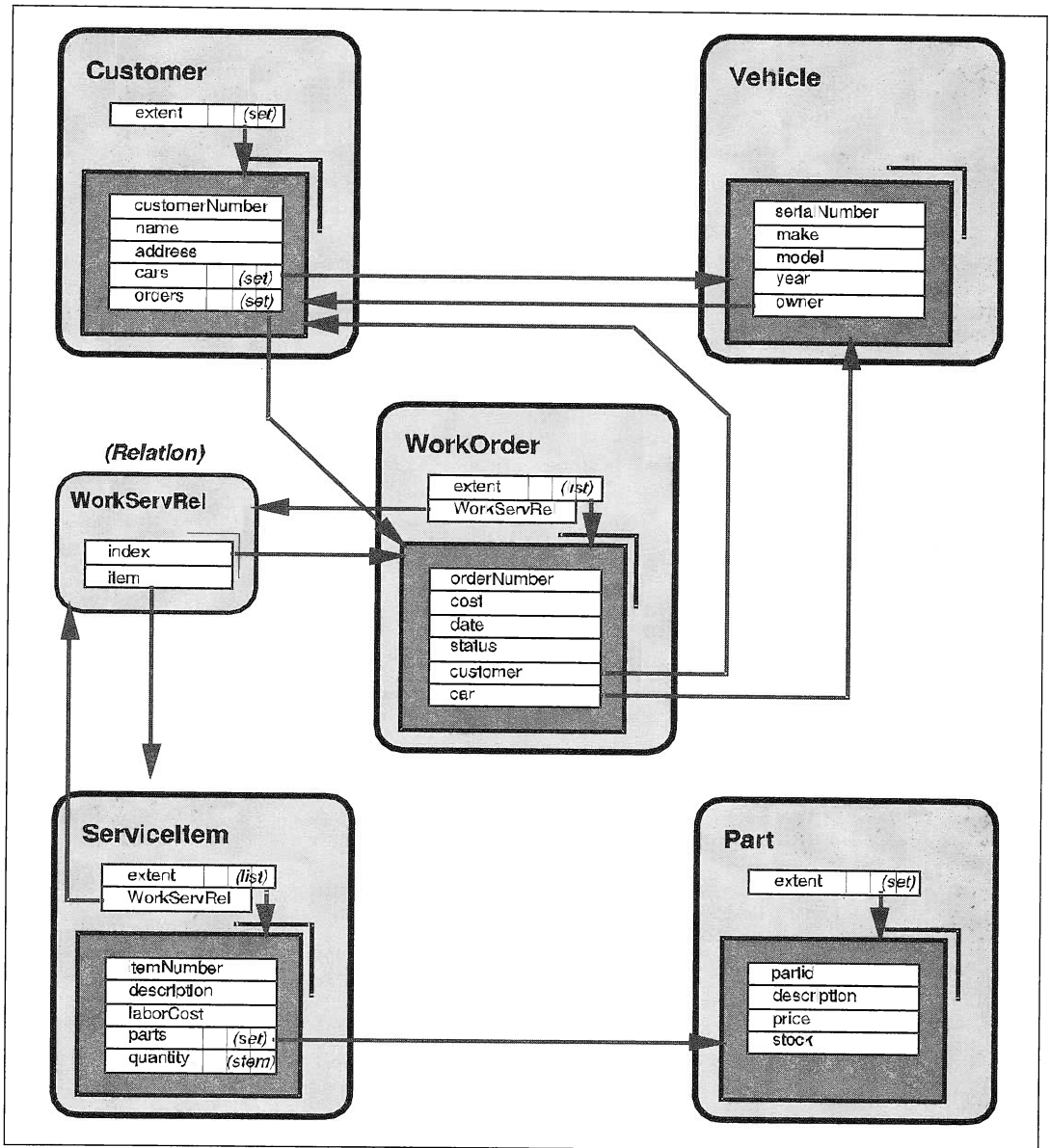


Figure 8. *Implementation of the Car Dealer Model*

The outer rounded boxes represent the classes, the inner rectangles the instances (objects) of the class. The white boxes in the outer rounded boxes are the attributes of the class; the white boxes in the inner rectangles are the attributes of the instances. Arrows indicate attributes that point to object instances.

Implementation Notes

1. We chose to use the set, list, and relation classes to experiment with the features of these collection classes. For work orders, for example, we chose to have a list so that new work orders are added at the top.
2. The relation class is well suited to implement the m:m relationship between work orders and services. It provides methods to get a list of related objects:

```
::class WorkOrder
::method addServiceItem
  use arg itemx
  workserv = self~class~getWorkServRel
  workserv[self] = itemx      /* add a service item to the work order */
::method getServices
  return self~class~getWorkServRel~allat(self)

::class ServiceItem
::method getWorkOrders
  return self~class~getWorkServRel~allindex(self)
```

The method *getWorkServRel* returns a pointer to the external relation object and the *allat* and *allindex* methods of the relation class return an array of related objects.

The relation object is implemented in the local directory (see *The Local Directory* on page 153) as

```
.local[Cardeal.WorkServRel] = .Relation~new
```

3. The methods to list all the objects of a class (*ListCustomerShort*, *ListPart*, etc.) are implemented as *routines* instead of methods. Object REXX provides the *::routine* directive to define subroutines (callable procedures):

```
::routine ListPart public
  aui~LineOut('List of' .Part~extent~items 'parts:')
  aui~LineOut(.Part~heading)
  do partx over .Part~extent
    aui~LineOut(partx~detail)
  end
  aui~EnterKey
  return
```

The decision to use routines is based on the assumption that this code is used only by the ASCII user interface and not by the GUI.

Sample Class Definition

Figure 9 shows an abbreviated listing of the *Customer* class as implemented in memory.

```

::class CustomerBase public

::method initialize class                                /*----- class methods -----*/
  expose extent
  extent = .set~new                                     /* prepare set of customers */
::method add class                                       /* add customer to set */
  expose extent
  use arg custx
  extent~put(custx)
::method remove class                                   /* remove customer from set */
  expose extent
  use arg custx
  extent~remove(custx)
::method findNumber class                               /* find customer by number */
  expose extent
  parse arg custnum
  do custx over extent                                  /* - look through the set */
    if custx~number = custnum then return custx
  end
  return .nil
::method findName class                                /* find customer by name */
  arg custsearch
  custnames = .list~new                                  /* - prepare result */
  do custx over self~extent                             /* - look through the set */
    if abbrev(translate(custx~name),custsearch) then do /* compare name */
      custstring = custx~number~right(3)|| ,
                  '- 'custx~name'- 'custx~address
      custnames~insert(custstring)                      /* - add one to result */
    end
  end
  return custnames~makearray                             /* - return the result */
::method extent class                                  /* return set of customers */
  expose extent
  return extent~makearray
::method heading class                                 /* return a heading */
  return 'Number      Name
          Address'
::method init                                           /*----- instance methods ----*/
  expose customerNumber name address cars orders
  use arg customerNumber, name, address                 /* initialize new customer */
  cars = .set~new
  orders = .set~new
  self~class~add(self)                                  /* add it to the set of cust*/
::method delete                                         /* delete a customer */
  expose cars orders
  do carx over cars                                     /* - and all the cars */
    carx~delete
  end
  do workx over orders                                  /* - and all the orders */
    workx~delete
  end
  self~class~remove(self)                               /* remove it from the set */

```

Figure 9. (Part 1 of 2) Customer Class in Memory

```

::method number unguarded                                /* return customer number */
  expose customerNumber
  return customerNumber

::method name attribute                                  /* name, name= methods */
::method address attribute                               /* address, address= methods*/

::method update                                          /* update customer info */
  expose name address
  use arg name, address

::method addVehicle                                    /* add vehicle to customer */
  expose cars
  use arg newcar
  cars~put(newcar)
  newcar~setowner(self)

::method removeVehicle                                  /* remove vehicle from cust */
  expose cars
  use arg oldcar
  oldcar~deleteOwner
  cars~remove(oldcar)

::method getVehicles                                    /* return vehicles of cust. */
  expose cars
  return cars~makearray

::method findVehicle                                    /* find vehicle by serial */
  expose cars
  use arg serial
  do carx over cars
    if carx~serial = serial then return carx
  end
  return .nil

::method addOrder                                        /* add order to customer */
  expose orders
  use arg newwork
  orders~put(newwork)

::method removeOrder                                    /* remove order from cust. */
  expose orders
  use arg oldwork
  orders~remove(oldwork)

::method getOrders                                      /* return all orders of cust*/
  expose orders
  return orders~makearray

::method detail                                          /* return a detail line */
  expose customerNumber name address
  return customerNumber~right(5) ' ' name~left(20) ' ' address~left(20)

::method makestring
  expose customerNumber name
  return 'Customer:' customerNumber name

```

Figure 9. (Part 2 of 2) Customer Class in Memory

Source Code for Base Class Implementation

The source code for the base implementation is described in Table 21 on page 255 and listed in *Base Classes* on page 293.

4

ASCII User Interface

In this chapter we look at a variety of technologies that can be used to develop the user interface for Object REXX applications. While most of the solutions that we present are graphical (GUI), we also present a simple ASCII character user interface (AUI).

Designing the User Interface

“Come on, Steve, you’re late for the meeting!” called Curt.

“I’m busy working,” Steve called back.

“You know that work is no excuse for missing a meeting, Steve,” responded Curt.

“Meetings are work, man,” grumbled Steve, gathering his ThinkPad and stopping to pour some coffee before joining Curt and Hanna in the meeting area.

“Whoops!” said Hanna. “Being late for a meeting is bad enough, but coming in late *with coffee* is a capital offense, Steve. You know the rules!”

“Yeah!” agreed Curt. “You get to buy the cookies for our mid-morning coffee break. Make mine a blueberry muffin, please.”

Steve shook his head. “Someone has to do the work while you guys sit around talking to each other, or nothing would ever get done.” he said. “I’ve been designing the user interface for the car dealer application.”

“What do you mean, ‘designing the interface?’” asked Curt. “You know that Trusty Trucks doesn’t want a GUI front end to the application. All their existing PC applications are character-based, and they want the car dealer app to look exactly the same. Building a character-based user interface is the easiest thing in the world. We don’t need to waste time designing it.”

“Yes, I know,” said Steve, shaking his head. “It’s amazing. They’ve just upgraded all their PCs from DOS to Warp Connect so they can get easy file-sharing and Internet access, and they still want DOS-style interfaces. Surely you could have sold them on the benefits of a good GUI, Mr. Ace Salesman?” He looked pointedly at Curt.

“Be realistic, Steve,” responded Curt. “They’ve got a lot of legacy DOS applications that they have to keep running. One of the main reasons they chose OS/2 is its excellent DOS compatibility. They don’t want to redevelop all their old apps with GUI front ends. In fact, they don’t even have source code for some of the older ones. They were built by a little contracting company that went out of business. Like we might, too, if we don’t get a move on with this project!”

“Exactly!” said Steve. “That’s why I was working on the user interface. And for your information, the reason we need to design it carefully is that we plan to sell the same application to other businesses, as well. You know that I’ve been talking to Classy Cars about it, and they’re really interested. There’s no way they will want a clunky old character interface app in their smart showrooms with the sort of cars they sell. They want the latest GUI, where the *G* stands for ‘Gee Whiz!’ And when I showed them some multimedia with bitmap displays, recorded audio, and even a short video clip, they were turning handsprings.”

“Multimedia? With audio and video?” snapped Curt. “Steve, are you crazy? This is a simple car dealer application. It handles the booking and tracking of vehicle services. We’re talking about guys in greasy overalls crawling under cars. Multimedia has absolutely nothing to offer us in this application. We’ve got a tight deadline to deliver working code to support a serious business operation, and you’re messing around with your multimedia toys again! You’re just trying to justify the company money you wasted buying that fancy multimedia Think-Pad.”

Steve smirked in reply. “Your trouble is you have absolutely no imagination,” he said to Curt. “Sure, we’re building a vehicle servicing application. But the reason we chose to build it in Object REXX is that, as time goes by, its OO facilities will allow us to reuse the objects we build, for totally new applications. And while I was talking with Classy Cars, I found that their real hot button is the sale of cars, both

new and secondhand. Sure, they need a system to manage car services, but they make more money selling cars. And when I talked through the selling process with them, I soon realized that our application already contains many of the objects needed to support selling. Like vehicles and customers, for example.”

“But Steve, we have no multimedia data in our system at all as it stands,” said Hanna. “And writing code to handle multimedia will be a major undertaking. Is it realistic to start with something that complicated at this stage?”

“What you guys don’t realize is how easy it is to do multimedia in REXX,” said Steve. “I developed a little demo on the fly at Classy Cars to show them how it could work. Just look at this.”

Steve double-clicked an icon on his ThinkPad’s desktop, and a folder opened. In it were several icons, each looking like a car, with captions referring to different popular brands. There were also icons labeled *Audio* and *Video* in the folder. Steve dragged one of the car icons and dropped it onto the Audio icon. After a second, they heard his recorded voice saying, “The Gazebo. An ideal outdoor car for the driver who enjoys plenty of fresh air.” Steve picked up the same icon again and dropped it onto the Video icon. A window opened on his screen, and the familiar scene of two macaws beaking one another played in it, accompanied by music.

“Is that all it does?” asked Curt. “And how many hundreds of lines of code did you waste building it?”

Hanna looked thoughtful as well. “How many lines of code did it take, Steve?” she asked.

“Hardly any,” said Steve. “Look, this is just a demo to show what could be done, not a production system. I used Warp’s standard WPS facilities to put together the folder and icons. I pointed to my `audio.cmd` in the audio button’s setup notebook. When the user drops an icon on Audio, my command gets scheduled with the name of the icon dropped on it as the parameter. I use this name to pick an audio file and play it. Here’s the code.”

Steve opened an editor window, and there, hiding in the top corner, were 10 lines of code. “The video works the same way,” he added, and brought up the `video.cmd` for them to see, as well.

Hanna was excited. “Hey, Steve, that’s really neat. If that’s what you were building just now, I’ll pay for the cookies in the next coffee break.”

“This is just a red herring,” said Curt. “We’ve got to deliver the car-servicing application soon if we want to get Trusty Truck’s business. And if we want to stay in business. We don’t have time to mess around building multimedia demos.”

Steve looked upset and was about to respond, but Hanna cut in. “Wait, Curt. You’re right that we have to deliver Trusty Trucks’ application soon. But Steve’s also right. We’re using Object REXX precisely so that we can extend the base application to do different things for different customers in the future. We have to strike the right balance. And the key to that is to design the system right from the start so it can grow to meet new needs as they arise.”

“And that was exactly what I was doing when you interrupted me to come to this meeting,” said Steve. “So why don’t we stop talking and get some work done?” He brought up his notes on his ThinkPad and started talking through them. “We will have to develop both character and GUI versions of our application. I’ve called the character version *AUI* for short. Most of the functions that we have to implement will be common to both versions, and, of course, we don’t want to duplicate the code.”

“Why not?” asked Curt. “That’s how we’ve always done it in the past. It’s a clean solution. You can implement a change that one customer wants without messing up the other customer’s versions.”

“Well, we didn’t have very much choice in the past,” said Steve. “Classic REXX is procedural, and it’s hard to share procedural code between different versions of an application. It’s even hard to segment a large application into many small files with classic REXX because of the communication barriers between different source files. And if the individual source files are big, it follows that they deliver complex functions. The key to reuse is keeping the functions small and simple. That’s where Object REXX shines.”

“Maybe Object REXX makes it easier to share code,” said Curt, “but what’s the advantage? Different customers will want different features added to the application, and it won’t be long before they’ll need different versions of the source.”

“We have to break out of that cycle if we want to be more profitable,” said Hanna. “If you look at the really successful PC software products, there isn’t a different version with different features for each customer. The vendor implements only those features that will be useful to most of the customers, then they *all* get the new features. We aren’t quite in that kind of business, and we will have to implement some features that only one customer requests. But we should always be on the lookout to make new features available to every customer who might possibly want them, even if it’s some time out in the future.”

“Yes, and Object REXX will allow us to keep a common code base in the form of common class definitions in a shared source file, and then to implement only the *differences* as source unique to a particular customer,” agreed Steve. “That’s why I’ve been trying to work out how we can implement the ASCII user interface as an object.”

ASCII User Interface As an Object

“Did I hear you right?” asked Curt. “You want to implement the ASCII user interface as an object?”

“That’s right, Curt,” replied Steve. “We’re using an object-oriented language, you know, so why not use its facilities for the user interface, too?”

“Well, I’ve got good news for you, Steve,” Curt said. “In Object REXX, every variable and expression is actually an object, and every function is a method on an object. When you code:

```
say aString
```

“that’s actually equivalent to:

```
call lineout aString
```

“and Object REXX implements that as:

```
.OUTPUT~lineout(aString)
```

“where *.OUTPUT* is a standard object in each process’s directory of local values. You need struggle no longer to design an OO interface for the AUI version, Steve. Just use the *SAY* command!”

“I was trying to work at a *slightly* higher level of abstraction than that,” said Steve. “Think about the menus, for example. Our old REXX programs that run in AUI mode are riddled with strings of *SAY* commands that spill menus out on the screen. Those won’t work so well when we have to switch the output from the default *.OUTPUT* object to a GUI screen driver, will they, Curt? We would probably want to make use of the GUI window’s menu bar instead,”

“That’s why we need different versions of the source for the AUI and GUI versions,” replied Curt.

“Can we put the AUI-handling code and menus into a subroutine in a separate file?” asked Hanna. “The GUI version would simply not call that subroutine and not have that code.”

“It’s not that easy,” said Steve. “The logic that handles the menu has to call the code that implements the menu option chosen by the user. The menu code has to be the main routine, and the rest of the system subroutines. If they’re in separate files, it’s not that easy to pass them all the data they need to run.”

“That’s why we need different versions of the source for the AUI and GUI versions,” parroted Curt.

“We’re trying to solve a problem, not score cheap points,” said Hanna. “The problem of communicating with subroutines isn’t that great with Object REXX—we can encapsulate all the data they need in a single object, if necessary. But tell us the approach you were working on, Steve.”

The AUI Class

“Well, we could implement an AUI class to handle all output to the console. Handling the screen-full condition is always a bit messy, and in the past we’ve written that logic into every piece of code that produces a listing on the screen. The AUI *init* routine would be called automatically when we create the AUI object, and it could use the REXX *SysTextScreenSize* built-in function to find out how many rows can fit on the screen. Each time the *lineout* method is used, it can call the REXX *SysCurPos* built-in function to find out how many screen rows are already used, and handle the screen-full condition automatically.”

“Sometimes we need to put out several related lines that we want to stay together on the screen—perhaps a customer with all of his or her vehicles,” said Hanna. “How would the AUI object handle that?”

“We could build a *checkRows* method for AUI,” replied Steve. “You pass it the number of lines you want together on the screen. It checks to see if there’s enough space free, and if not, it invokes the screen-clearing logic. With this approach, none of the routines that generate screen output would need to know how many lines the screen has space for, or how many of its lines are already in use. All of this screen-related information would be *encapsulated* in the AUI object.”

“That’s neat,” said Hanna.

The AUI Operations

“Let’s define all the operations (methods) needed for the character interface on a sheet of paper,” added Curt. Shortly afterward, they had it all ready [see Table 11].

Table 11. Methods Required for AUI		
Method	Type	Purpose
init	Instance	Initialize object, store window size
getrows	Instance	Return number of rows in window
ClearScreen	Instance	Clear output window
LineOut	Instance	Output one line of text
CheckRows	Instance	Check if there is enough space for “n” rows
UserInput	Instance	Ask user for input, character or numeric
YesNo	Instance	Ask user for Yes or No input
Enterkey	Instance	Wait until user presses enter key
Error	Instance	Display error message
AckMessage	Instance	Display acknowledgment message

“Why are the methods instance methods?” questioned Curt.

“Well, Curt, it’s true we could implement all methods as class methods,” replied Steve, “but I think it is cleaner to create an actual *AUI* object at run time to handle the interactions.”

ASCII Menus as Objects

“The other things I’d like to tidy up are the menus. In the past we have implemented them by coding a whole bunch of *SAY* instructions, outlining the options. Immediately after these come *WHEN* instructions, checking for each of the options and implementing some action if that option has been chosen. I’d like to move the menu text and associated actions right out of the REXX programs and store them as parameter files. Then we could define a menu class. Its class *init* method could read the menu parameter file and set up the menus as a list of objects in storage. It would also have a method to display a selected menu object, check which option the user chooses, and automatically implement the corresponding action.”

“Often, a menu action will simply display a submenu,” said Curt.

“That’s right,” said Hanna. “In that case, the action in the main menu would be an instruction to display the submenu. The menu display method would invoke itself. That’s possible, isn’t it?”

“Yes,” said Curt. “I’ve been doing some playing around with the concurrency features of Object REXX...”

“So I’m not the only one who plays around!” interrupted Steve.

“...and methods can invoke themselves recursively,” continued Curt. “Of course, we would have to make sure we don’t send them into an infinite loop by linking a submenu back to one of its parent menus.”

“Our menu display process would be user-driven. The user would quit soon enough if it loops back on itself,” said Steve.

“That sounds like a good approach,” said Hanna. “I’m still not clear on how we’ll tie it all together when we build our first GUI front end, but that’s not today’s problem. It’s time for our coffee break. I’ll buy the cookies.”

“Great!” said Steve. “In that case, let’s go to the Golden West Coffee Shoppe.”

“Oh no!” groaned Curt. “Don’t tell me that I have to sit and watch him nibble his way through yet another Golden West Monster Munch Chocolate Chip Cookie!”

The Menu Operations

After the coffee break, the three sat together and developed the list of menu operations [see Table 12].

Table 12. Methods Required for Menu

Method	Type	Purpose
initialize	Class	Read menu file and build menu objects
findMenu	Class	Find existing menu or allocate new one
init	Instance	Initialize new menu object with array of items
addItem	Instance	Add a menu item to the menu object
getname	Instance	Return name of a menu object
showMenu	Instance	Display the menu, prompt user

Implementing the Menus

The menu input file MENU.DAT has the following structure, with fields separated by tab characters (represented as ~ signs):

Structure of menu data file

```

Main~CAR DEALER - GENERAL MENU
Main~List (customer, part, work order, service)~showMenu List
Main~Update customer and part information ~showMenu Update
...
List~CAR DEALER - LISTING MENU
List~List customers ~call ListCustomerShort
List~List customers and vehicles ~call ListCustomerLong
...
Update~CAR DEALER - UPDATE MENU
Update~Create a new customer ~call Newcust
...

```

The main program uses the menu class as follows to initialize the menu structure and to run the application using the menus:

Menu loop in main program

```

au = .AUI~new /* allocate AUI object */
menus=.array~new /* runtime level array of menus */
menus[1] = .Menu~initialize /* build menu objects, store 1st */
level = 1 /* start at top menu */
do until level < 1 /* run loop until exit */
  action = menus[level]~showMenu /* show the current menu */
  select /* - user enters an action */
    when action = .nil then level = level - 1 /* previous menu */
    when action~class = .Menu then do /* user select submenu */
      level = level + 1 /* - add submenu at lower level */
      menus[level] = action
    end
    otherwise interpret action /* user select an action */
  end /* - run that action */
end

```


Appearance of ASCII User Interface

The windows displayed to the user by the menu system are shown in Figure 10 on page 66.

Source Code for ASCII User Interface

The source code of the AUI and menu implementation is not listed in the appendix; it is available in the car dealer directory on the CD or after the sample applications have been installed on a hard drive (see Table 25 on page 256).

The source code to start the AUI program is listed in *Program to Run the Car Dealer Application* on page 327.

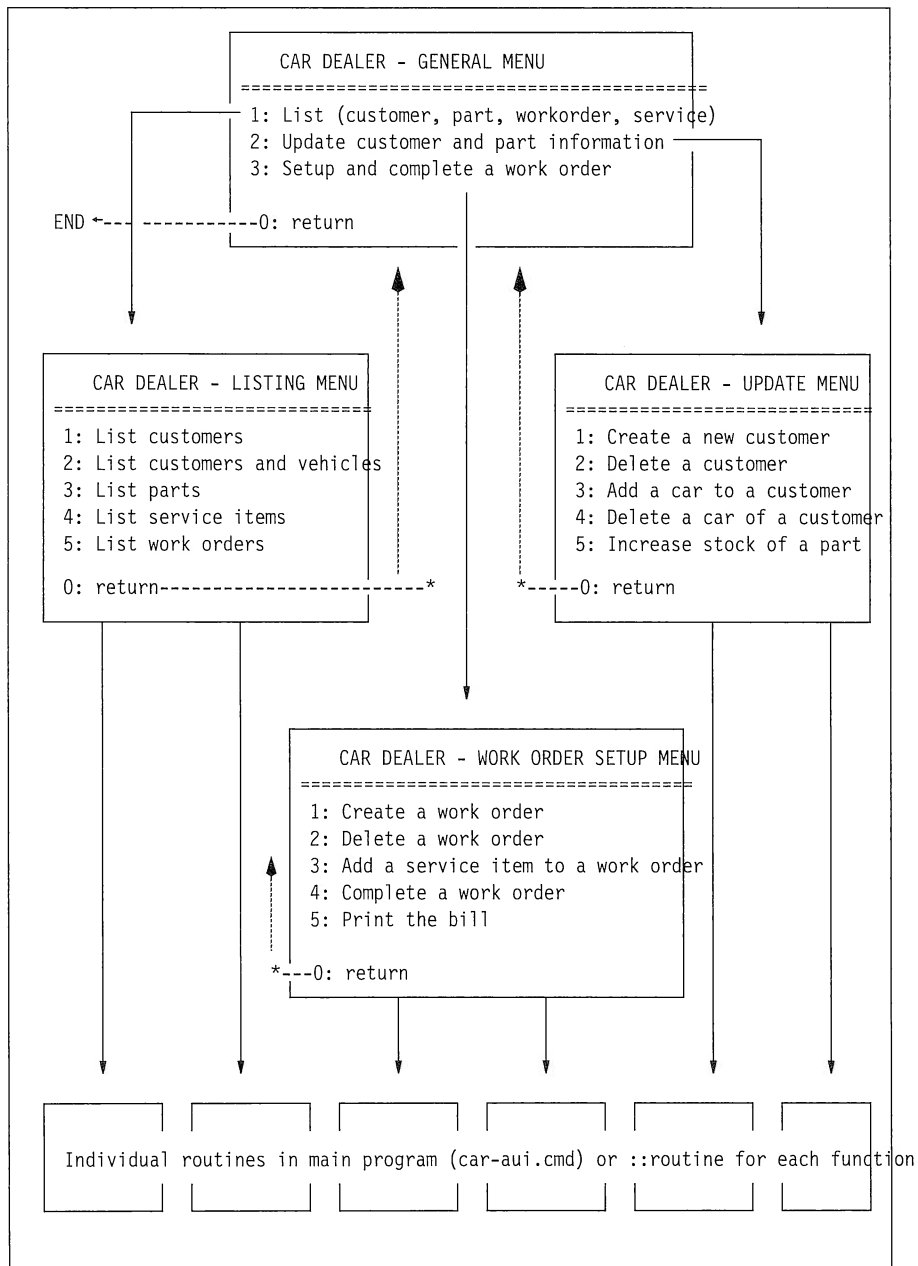


Figure 10. *Appearance of ASCII User Interface*

5

Persistent Objects on Disk

In this chapter, we find out how objects can be made persistent by storing them in conventional flat ASCII files. We want to ensure that the objects survive (even when the program that creates them ends) and are available again the next time that the program runs. For brevity, we call this the *FAT* (File Allocation Table) *option*, although, of course, the files may equally well be located on a High-Performance File System (HPFS) drive.

Storing Objects in FAT Files

“Great news, team—we’re on the last lap of agreeing on the detailed design of the car dealer system with Trusty Trucks!” Curt strode into the Hacurs premises, stripping off his coat and beating a light dusting of snow from it.

“Wonderful!” said Hanna. “We’ve all spent a lot of time on this application. We need to implement it and bill for it soon.”

“The only outstanding area is, How are we going to store their objects when they turn off the PC that runs the application?” asked Curt.

“Did you say ‘the PC?’” asked Steve. “Aren’t they going to run it on multiple machines?”

“No,” replied Curt. “We did some performance evaluation with our latest prototype and worked out that they’ll be able to handle their current business volumes easily on a single workstation. And Trusty Trucks is too cost-conscious to use two PCs to do the job if one will do.”

“What if the one PC breaks down?” asked Steve.

“Well, that’s part of what we have to discuss and implement,” replied Curt. “We need to make sure that all object creations, updates, and deletes are recorded on disk, and that the system will recover its objects automatically from disk when it is brought up again. We also have to decide on some kind of disaster recovery scheme.”

“The obvious way to do this is to base our object persistence on a database manager like DB2,” said Steve.

“Trusty Trucks is a cost-conscious organization, Steve,” replied Curt. “They’re not like Classy Cars. They don’t own a real database management system, and they’re not about to buy one just to run our application. We’re going to have to find a way of doing the job using conventional ASCII files, if that’s possible.”

“If only a single workstation needs to access the data, it’s entirely possible to do the whole job using ASCII disk files,” said Hanna. “In fact, that approach would work even if multiple PCs want to read the files at the same time, so long as only one PC has update access to the files.”

“That would be the ideal solution, Hanna,” said Curt. “How would you go about it?”

“I don’t think I’m going to like this!” interjected Steve, but the other two ignored him.

Hanna swept her hair back with her fingers, walked to the whiteboard and picked up a pen. Curt and Steve knew she wouldn’t use the pen, just holding it seemed to help her think. “Because we’re using an OO approach to the whole system, we know exactly when a new object is created, or an old one updated or deleted. It can’t happen unless our object methods are called.”

“Right,” said Steve and Curt in unison.

“OK. So all we have to do is change the *init* method for each object to write a copy of the new object to disk as soon as it has finished initializing it. And we make similar changes to the delete methods so they can delete the objects from disk, and to the update methods so they can rewrite the objects to disk whenever they change.”

“That sounds straightforward,” said Curt.

“How would you delete an object from a disk file if it’s in the middle of the file?” asked Steve. “You can’t just leave a hole in the file.”

“We could shift all the trailing objects one place to the left and leave out the deleted object,” replied Hanna.

“How will we know the position of the object in the file?” asked Steve. “We can’t shift the remaining objects over it unless we know where it is.”

“We could give every object a new attribute called *position*,” suggested Curt. “When we read the objects into storage, we could store the positions in which we found them in this attribute. Then when we need to update or delete them, we’ll know where they are on disk.”

“But if they keep shifting around every time you delete an object, you won’t know where they are when it comes time to update them,” reasoned Steve. “It would really be simpler if we used a database manager. It has all the logic needed to sort out problems of this kind.”

“I’ve just told you, Steve...” started Curt, but Hanna interrupted.

“Hold on, guys! Curt, how many objects are there going to be?”

Curt shifted his scowl from Steve and answered Hanna. “A couple of hundred vehicles, somewhat fewer customers, and maybe a thousand service records if they keep six months’ history on line.”

“So why don’t we just rewrite all the objects to disk any time one of them is changed?” suggested Hanna.

“That would be a huge overhead!” Steve objected.

“Not necessarily,” answered Hanna. “We need to store only 30 to 40 bytes of information per object, and if there are a thousand, we have to write 30 or 40 KB. That won’t take long.”

“It would take forever!” said Steve. Curt began tapping on this Think-Pad’s keyboard. “Anyhow,” continued Steve, “if you add up all the objects of all types, it comes to a pretty big file, maybe 120 KB.”

“No, we would split the different object types into separate files,” said Hanna. “Otherwise we’d have a jumble of different object types in the file, and we’d have to write extra logic to separate them. So the biggest file we’d have to handle would be only about 40 KB.”

“That would still take a long time to rewrite,” said Steve.

“Let’s time it,” said Curt. “I’ve just written a little REXX command that takes two parameters—average record length and number of records. It writes a file of this size to disk, and measures how long it takes. So what numbers should I try?” he asked.

“Try a 30-byte record length and 1000 records,” said Hanna.

“OK,” said Curt, and he typed in the command. “That took one second,” he said. “Doesn’t sound too long to me,” he added, looking at Steve.

Steve looked stunned for a while, then his bewildered look cleared up. “Of course! You’re using an HPFS-formatted disk. That gives you high performance to start with, and you’ve probably got ‘lazy write’ on,

too—that's the default for HPFS. So OS/2 will accept the write commands into buffer and come back to the program immediately. The data gets written back to disk in parallel with subsequent program execution."

"That sounds great to me," said Curt. "Since the operating system takes care of that for us, we really don't have to worry about performance. And the production system will run on a server at Trusty Trucks. Its disk is faster than mine. So Hanna's approach will perform beautifully."

"Yes, but what if the system goes down while it's still writing to disk?" asked Steve. "A database manager logs all changes to disk, as well as writing the data back, so if something goes wrong while it's writing the data, it can always fix it up from the log."

"The data files for Trusty Trucks will be so small that we could easily write the data out to a different file name each time around, and cycle through the list of three or so file names for each file," said Hanna. "When we start up the system, we can use the *query timestamp* operand of the stream command to find out which version of the data is the most recent. We can write a special trailer at the end of each file, so if it's not there we know the file is incomplete and we should use the next most recent one. It's easy to solve this kind of problem."

"I agree," said Curt. "Let's get the system going against flat ASCII files and start doing user training and acceptance testing at Trusty Trucks. We can change it later to put in smart recovery logic. The users are waiting for us, and we need the money we'll get once this system is installed."

"That's fine for Trusty Trucks," said Steve, "but Classy Cars is a much bigger operation. They need to support six to eight separate locations, and they will need update facilities from multiple workstations at the same time. There's no way a simple ASCII file approach will meet their needs."

"You're absolutely right, Steve," agreed Hanna. "We'll have to build database support when it's time to implement the system for Classy Cars. And maybe in the past we would have had to try to persuade Trusty Trucks to use a database manager as well, because we couldn't afford to support two different versions of the application. But the main reason we're using Object REXX for this application is so we can customize different versions for different customers and still reuse all the common business logic. Remember?"

"Yes, I guess so," said Steve, not looking convinced. "Except that once we've got the ASCII file version going, you'll probably change your minds and decide that's the way we have to go for Classy Cars too."

"We'll do what's best for the customer, Steve," said Hanna. "That's the only way to make sure that they'll ask us for help again."

“OK,” he said, “We’ll build an ASCII file solution for Trusty Trucks, but I’m going to start working out how we can structure the application to get the kind of configuration flexibility we’ll need in the future.”

“That’s a great idea, Steve,” said Hanna. “Now, let’s finish this design and get coding. What format should we use for the objects when we write them out to disk files?”

Format of the Objects

“Why not write them out as comma-delimited records, the way a spreadsheet package would export rows?” asked Curt.

“Not bad, but our data could easily include commas in things like address fields,” said Hanna. “Let’s use tab characters to delimit fields.”

“Good idea!” said Curt. “I’ve got the fields we need on this piece of paper.” He rifled through his work file and pulled out a piece of paper [see Figure 7 on page 42]. “All we have to do is write the files in this format.”

“Just a minute,” said Steve. “You’ve got some *smallint* fields defined there. Those are 2-byte binary integers. Either byte could easily contain the code value for a tab, the ASCII value 9. That would throw you off when you try to decode the record during loading.”

“Good thinking, Steve,” said Hanna. “We’ll have to write the numeric values as strings. No problem, because that’s the way REXX writes them out, unless you tell it otherwise.”

Sample ASCII file for the customer class

```
number name          address          (↵ = tab key)
101  ↵Senator, Dale   ↵Washington
102  ↵Akropolis, Ida ↵Athens
103  ↵Dolcevita, Felicia ↵Rome
...
```

Steve smiled his appreciation for Hanna’s compliment and got more enthusiastic about the design. “We’ve also got some repeating groups in the service and work order objects,” he said. “How are we going to handle those?”

“We can just attach them to the back end of the record, delimited by tabs, like the other fields,” said Curt. “We’ll know what they are when we read the files. There’s no chance of confusion.”

Sample ASCII file for the work order class

number	date	cost	complete	custmr	serial	service-items (↵ = tab key)
1	↵09/06/95↵↵1	↵0	↵101	↵123456	↵1	
2	↵09/06/95↵↵1	↵0	↵103	↵398674	↵10↵9↵4	
3	↵09/06/95↵↵1	↵0	↵106	↵911911	↵7↵6	
...						

“Not exactly third normal form¹!” said Steve.

“No problem,” said Hanna. “No one is going to read these files except us, while we’re debugging the code.”

“Hmm—I guess so,” agreed Steve reluctantly.

Implementing the Changes in Code

“Well, I think we’ve got this all sorted out,” said Curt. “We just have to modify the methods we wrote for our objects, to write updates to disk...”

“Hold it!” snapped Steve. “We’ve just agreed that we’re going to have different versions of the system supporting both ASCII files and a database manager, and you want to start carving up our existing methods to hard-wire ASCII file logic into them. Once we’ve done that, we’ll *never* be able to support two different versions while sharing common code.”

“Be realistic, Steve!” said Curt. “How are we going to support ASCII files if we don’t write some new function into the code? This OO stuff isn’t magic, you know.”

Steve glared at Curt, then strode to the whiteboard and took the pen from Hanna. She sat down, grateful to give her feet a rest.

“Let’s start with customers,” he said. “We already have a *Customer* class defined with its methods to control how customer objects behave once they’re in storage.”

“Right,” said Hanna encouragingly, as Steve drew a box and labeled it *Customer* on the board.

“Currently, the *Customer* class is a subclass of the *Object* class by default, since we didn’t say otherwise when we defined it.” Steve drew a box labeled *Object* above the *Customer* box and connected the two. “Now, let’s say we change the name of the *Customer* class to *CustomerBase*, and define a new class called *FAT Customer*.” Steve drew a new box with this label below the *Customer* box, and drew connecting lines to show that *CustomerBase* was a child of *Object* and *FAT Customer* a child of *CustomerBase*.

¹ Third normal form does not allow repeating fields within a table. A separate table with a row for each repeating value is used in normalized tables.

“With this structure,” Steve added, “we could write the additional methods that we need for object persistence in FAT files as new methods in the *FAT Customer* class. We could change the *CustomerBase* class methods to invoke these new persistent methods when object updates need to get written to disk.”

“Hold on,” said Curt, “*FAT Customer* isn’t a valid class name.”

“Well, the class name would actually be just *Customer*,” Steve answered, “but we would have different versions of the *Customer* class definition; one for FAT, another for DB2. We would store them in separate files. I thought of giving both files the same name, but with different extensions to distinguish them—maybe *customer.FAT* and *customer.DB2*, for example.”

“That looks fine, Steve,” said Hanna, “but how would we switch between the FAT and DB2 versions of the code?”

“Like this,” said Steve. He drew a *DB2 Customer* box next to the *FAT Customer* box, and then changed the lines [Figure 11].

“Let’s suppose we need a DB2 version,” he said. “We develop a totally separate class called *Customer* with its own methods to handle persistent storage in DB2 tables. We set up two different configurations. In the FAT configuration, the *FAT Customer* class inherits its persistent methods from the *CustomerBase* class, and in the DB2 configuration the *DB2 Customer* class inherits them from the *CustomerBase* class. With this approach we can use exactly the same class and method definitions for both the FAT and the DB2 implementations.”

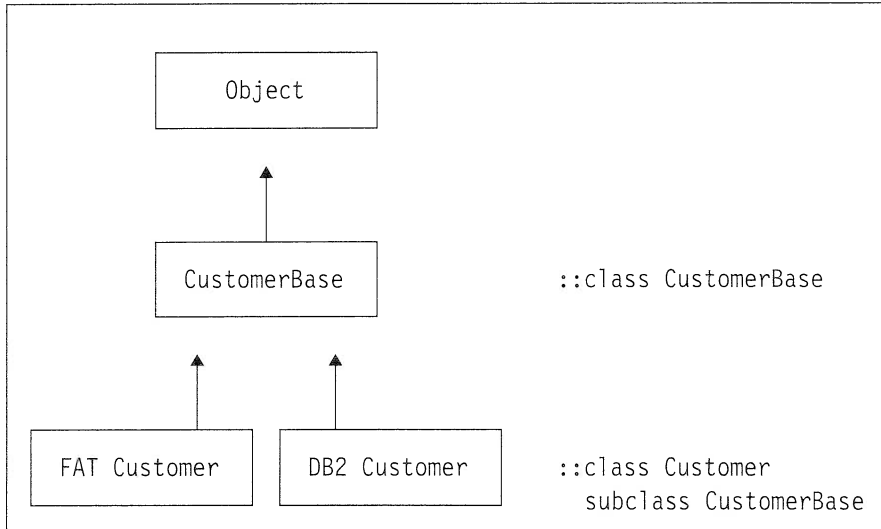


Figure 11. *Customer Class Inheritance Diagram*

“That’s really neat, Steve,” said Hanna. “That’s using the power of inheritance in Object REXX to solve a real problem. What do you think, Curt?” she asked, turning to him.

“It sounds like it might work,” said Curt, “but I think we should think it through a bit more before we decide on it.”

Hanna turned back to Steve. “Why don’t you try coding-up some sample code, Steve?” she said. “I’ll gladly help you if you need me.”

“I’ll rough something out for *Customer*, and we can get together again and check it out,” he replied. “If it works out the way we need, we can split the remaining classes between us and make corresponding changes to them.”

“OK,” said Hanna. “Do you think you’ll have something ready for us to look at tomorrow?”

“Is that a question or an order?” asked Steve. “I think I’ll have something for you to look at. After all, when you’ve seen one baseball game, you’ve seen them all.”

“Oh Steve, you make me feel terrible!” said Hanna.

“But not as terrible as you’ll feel if you don’t deliver, Steve!” added Curt. And on this note they parted.

The Class Structure

The next morning, Steve was in early. When the other two arrived, Steve called them over to his ThinkPad. He had plugged it into the big screen so they could see what he was doing while he worked with the smaller screen of the ThinkPad.

“This turned out easier to do than I expected,” said Steve. “I actually had time to watch the game. Too bad about the result! I’ve defined the new FAT Customer class to handle persistent storage on ASCII disk. It has only six methods.” He pointed to the screen, which showed:

- ❑ A *persistentLoad* class method to load all customer objects from disk into storage when the system comes up:

```
::method persistentLoad class
  expose file
  file = 'customer.dat'
  call stream file, 'c', 'open read'
  do i = 0 by 1 while lines(file)
    parse value linein(file) with customerNumber '9'x name '9'x address
    self~new(strip(customerNumber), strip(name), strip(address))
  end
  call stream file, 'c', 'close'
  return i
```

- ❑ A *persistentStore* class method to write all customer objects from storage to disk when any customer object changes:

```
::method persistentStore class
  expose file
  call stream file, 'c', 'open write replace'
  do custx over self~extent
    x = lineout(file, custx~fileFormat)
```

```
end
call stream file, 'c', 'close'
return 0
```

- ❑ Three methods—*persistentInsert*, *persistentUpdate*, and *persistentDelete*—that simply invoke the *persistentStore* method; for example:

```
::method persistentInsert
  return self~class~persistentStore
```

- ❑ A *fileFormat* method to convert the customer object into a tab-delimited string to be written to disk:

```
::method fileFormat
  return strip(self~number)'9'x || left(self~name,20)'9'x || ,
    left(self~address,20)
```

“There are only 45 lines of code in this class; it’s really simple,” Steve continued, showing them the code.¹

“What did you have to do to the original Customer class?” asked Curt.

“A few things,” replied Steve, and explained:

- ❑ “I had to change the *initialize* class method to invoke *persistentLoad*. This is the method in the *FAT Customer* class that loads all the customer objects from disk into RAM:

```
::method initialize class
  expose extent
  extent = .set~new
  self~persistentLoad
```

- ❑ “Then I changed the *init* method to invoke *persistentInsert* for a new customer:

```
::method init
  expose customerNumber name address cars orders
  use arg customerNumber, name, address
  cars = .set~new
  orders = .set~new
  self~class~add(self)
  if arg() = 4 then self~persistentInsert
```

- ❑ “And I changed the *update* method to invoke *persistentUpdate*:

```
::method update
  expose name address
  use arg name, address
  self~persistentUpdate
```

¹ The source code referred to by Steve is not included in this document. His application structure turned out to have some problems when Hacurs needed to introduce support for DB2. The *persistentInsert*, *persistentUpdate* and *persistentDelete* methods were moved to a separate mixin class called *Persistent*, as described in *The Persistent Class* on page 77.

□ “Last, I changed the *delete* method to invoke *persistentDelete*:

```
::method delete
  expose customerNumber name address cars orders
  do carx over cars
    carx~delete
  end
  do workx over orders
    workx~delete
  end
  self~class~remove(self)
  self~persistentDelete
```

“And that’s all I had to do. It was very simple, really,” said Steve.

“I don’t get it,” said Curt. “Why did you define separate *persistentInsert*, *persistentUpdate*, and *persistentDelete* methods if all of them simply invoke the *persistentStore* method? That looks like a waste of time.”

“Well, I’m thinking ahead to how we’ll implement the DB2 support,” said Steve. “With DB2 we’ll use different SQL commands to handle the insert, update, and delete cases. As it happens, we plan to handle these different cases by rewriting all the customer objects to an ASCII file for Trusty Trucks. But we need to invoke different methods in the base customer class so that one version can meet both requirements.”

“That’s good thinking, Steve,” said Hanna.

“Yes,” said Curt, “so long as he doesn’t waste too much time thinking about the DB2 implementation. We’ve got to deliver this system fast.”

“I’ve done my bit,” said Steve. “Why don’t you and Hanna make the corresponding changes for the other classes? Those are the *Vehicle*, *Service Item*, *Part*, and *Work Order* classes. You can use my code as the basis for your changes. I’ve put it on the server in the project directory.”

“Great! Let’s go.” said Hanna.

The Requires Directive

“Wait!” said Curt. “Are you planning to put the class definitions for customer, vehicle, service item, part, and work order into different files?”

“Yes,” said Steve, “it makes for a nice clean implementation. None of the class files will be more than a few hundred lines of code.”

“That may be so,” Curt replied, “but these classes refer to one another extensively. If you put them into separate files, will they still be able to use one another?”

“That’s why Object REXX has a `::requires` directive, Curt,” Hanna said. “It allows the code in one file to use classes and methods defined in another. As you go through the code, make a note of the classes it refers to, and just make sure that you include a `::requires` directive for each class that isn’t defined in the same source file.”

Example of `::requires` directive

In the *FAT Customer* class we require the definition of the *Customer-Base* class:

```
::requires 'carcust.cls'
```

The Persistent Class

Hanna and Curt settled down at their ThinkPads and started looking at the code. Pretty soon, Hanna called out, “Say Steve, we’re all going to code exactly the same three *persistentInsert*, *persistentUpdate*, and *persistentDelete* methods for each of the other four classes. Isn’t there a better way of doing that?”

“Come on Hanna, just cut and paste,” said Curt. “It won’t take long.”

“But Curt,” said Hanna “We talked about several different ways of implementing persistent storage and settled on our current approach for Trusty Trucks only because their file sizes are going to be small. What happens if their volumes grow, or if we find an opportunity to sell this solution to a bigger business that still wants a flat file solution but has large volumes?”

“Now you’re thinking OO, Hanna!” agreed Steve. “There is an easy way to do what you’re asking. I’ll move the three methods out of the *FAT Customer* class and put them into a new class—let’s say we call it *Persistent*. Then each of our five FAT classes can inherit these methods from the *Persistent* class.”

“So we should make our FAT classes subclasses of the *Persistent* class?” asked Curt.

“Yes. No! Wait!!” said Steve. “Let’s do this properly. I’ll make the *Persistent* class a *mixin* class. When you define your FAT classes, use the *inherit* clause to inherit methods from my *Persistent* class. That’s what mixin classes are for, after all. It’s very simple, really. Look, I’ll change the diagram to show how it would work.”

Steve drew shadow boxes behind the *Customer* class to represent the other data classes to be handled—*Vehicle*, *Work Order*, *Service Item*, and *Part*. He then added a new *Persistent* mixin class and showed that the FAT data classes inherited some methods from it [see Figure 12].

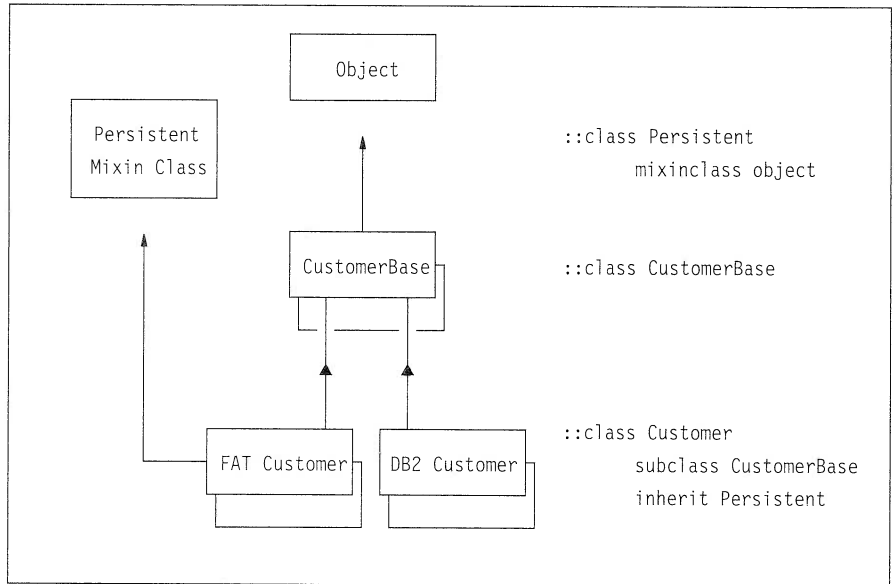


Figure 12. *FAT Data Classes Inheriting from a Mixin Class*

“What about the persistence methods for the DB2 version, Steve?” asked Hanna.

“The methods for persistence in DB2 will have to contain specific SQL statements for each object type,” replied Steve. “We’ll probably end up coding them directly into the DB2 classes themselves.”

“Is all this messing about really worth the trouble?” asked Curt.

“If we start building it right, we’ll finish building it easily,” said Hanna. “If we start cutting corners while we’re busy laying the foundations, there’s no way we’re going to get the walls square later on.”

Sample code of the Persistent class

```

::class Persistent public mixinclass Object

::method persistentLoad class          /* class methods */
  return 0
::method persistentStore class
  return 0

::method persistentInsert               /* instance methods */
  return self~class~persistentStore
::method persistentDelete
  return self~class~persistentStore
::method persistentUpdate
  return self~class~persistentStore
  
```

They all settled down to work. Their car dealer application was tested and working off persistent FAT file storage before the end of the day.

“I’m going to take this round to Trusty Trucks first thing tomorrow,” said Curt. “Wish me luck—I may just come back with the specs signed off and a committed implementation plan.”

“I’ll bring in a bucket of ice and some sparkling wine,” said Hanna. “Now don’t disappoint us, or Steve and I will have to drown our sorrows—alone.”

“You’ve got a deal!” said Curt.

Source Code and Sample Data for FAT Class Implementation

The source code for the *FAT* classes is described in Table 22 on page 255 and listed in *Persistence in Files* on page 310.

Sample flat files for the five classes of the car dealer application are described in Table 22 on page 255 (data subdirectory) and listed in *Sample Data* on page 289.

The source code of the *Persistent* class is listed in *Persistent Class* on page 308.

6

Graphical User Interfaces

In this chapter, we look at a variety of tools that can be used to develop a GUI for the Object REXX car dealer application. These tools are:

Dr. Dialog

VisPro/REXX

Watcom VX•REXX

The Setup

“That was some party!” said Steve. “Do you still have the Trusty Trucks contract, Curt? I hope you didn’t turn it into a paper plane.”

Curt smiled. “No danger of that, Steve” he said. “This contract is going to pay our salaries for the next few months while I’m busy installing the system and training the users.”

“The Trusty Trucks order was a wonderful business win, Curt” said Hanna. “It makes me feel really confident about the future of our company.”

“While you’re busy skinning this lion, I’ll go out and catch another,” said Steve. “Classy Cars is very keen on our car dealer application, and they’ll feel a lot more confident once they know they won’t be the only ones using it. The only thing is, I’ll have to develop a GUI front end for it.”

“It would be great if we could sell our system to other customers,” said Hanna. “That’s the way to get our profits up. We spent a lot of time designing it so that we could easily customize it to different users’ requirements. It would be a great shame if that were never put to the test.”

“Well, I’m going to start building the GUI today” said Steve. “I’ll let you know how easily it fits in with our current design.”

“Which GUI builder are you going to use, Steve?” asked Hanna.

“Dr. Dialog,” answered Steve. “That’s the one I know and love.”

“Good luck!” said Hanna. Steve smiled in reply, then started working on his ThinkPad.

The Car Dealer GUI

Two days later, Steve called to Hanna. “Would you like to see the new, improved, GUI version of the car dealer application?” he asked.

“I’d love to,” Hanna answered, pulling a chair across to his desk.

Steve double-clicked a Dr. Dialog resource icon labeled `car-gui.res` and a GUI window opened [see Figure 13 on page 83].

“This is the main window, and the users can do a lot with just this. The first thing they have to do is identify a customer. They can put a name, or part of a name, in the *Search for...* entry field, then hit the **Search** button. I fetch all customers whose names match the search pattern entered, and put them in the list box to the right.”

“And if they enter no characters at all in the search field?” asked Hanna.

“Then the customer *findName* class method will fetch all the customers,” answered Steve, demonstrating this as he spoke. “Now, if the user clicks on a particular customer in the list box, I fetch that customer’s details—number, name, and address—and display them in the entry fields below.” Steve clicked on a name, and the details were filled in.

“I also fetch a list of the cars owned by that customer and put them into the vehicles list box” added Steve. “If the user clicks on a particular car in this list, I fetch that car’s details and populate the *Vehicle* entry fields. And if the customer has only one car in our database, which will happen often, I select that car and fetch its details automatically.”

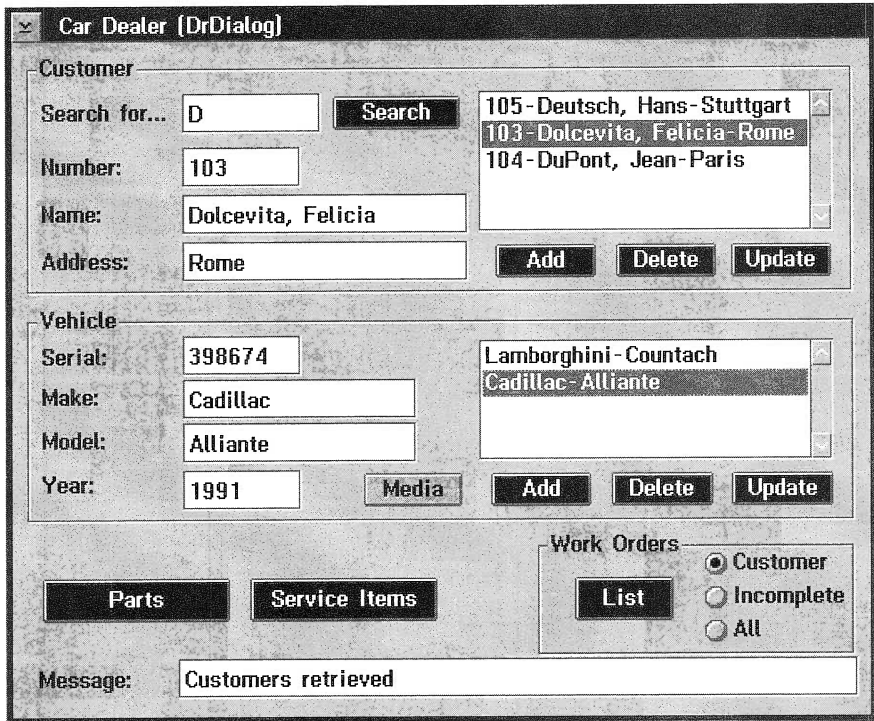


Figure 13. Main Window of Dr. Dialog GUI Application

“That’s neat,” said Hanna.

“I’ve got **Add**, **Delete**, and **Update** buttons under the customer list box,” Steve continued.

- ❑ “To add a new customer, the user fills in the customer’s number, name, and address, then clicks on the **Add** button.
- ❑ “To update a customer, the user selects the customer, overtypes the name or address, then clicks on the **Update** button.
- ❑ “To delete a customer, the user selects the customer and clicks on the **Delete** button.

“The **Add**, **Delete**, and **Update** buttons under the vehicle list box do similar things. And I’ve put a **Media** button there, too. It doesn’t do anything yet—it’s just a reminder.

“If the user clicks on the **Parts** button, I open a Part List window,” he continued, showing this happen as he spoke [see Figure 14].

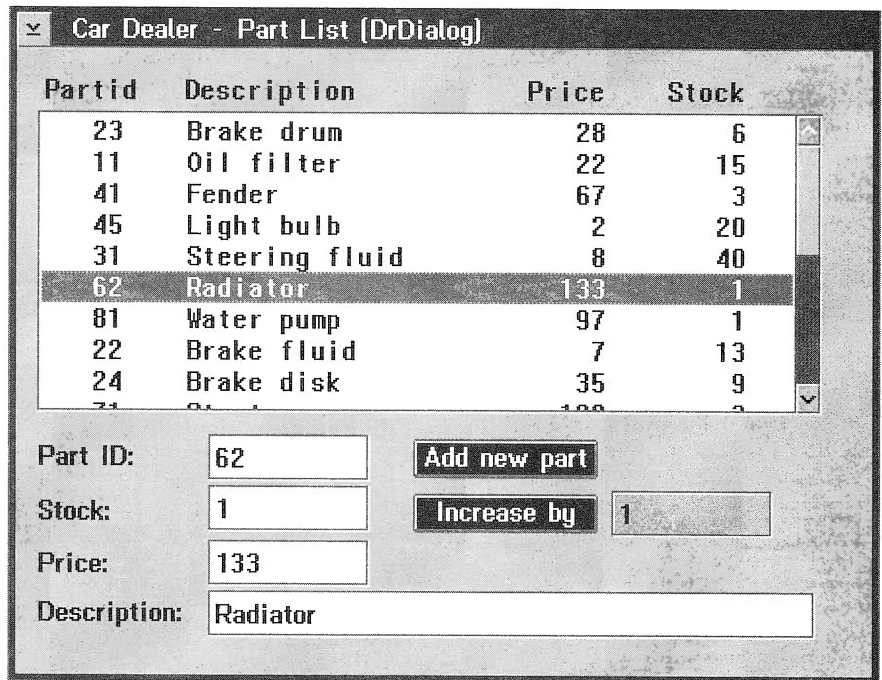


Figure 14. *Part List Window of Dr. Dialog GUI Application*

“This shows all the part objects available in a list box. If the user selects a particular part, I fetch its details and populate the entry fields at the bottom. The user can increase the *Stock* field by typing in the amount to increase by, and clicking on the **Increase by** button. To add a new part, the user fills in the entry fields and then clicks on the **Add new part** button.” Steve exercised these options.

“This looks great, Steve,” said Hanna.

“Thanks,” said Steve. He closed the Part List window. “If the user comes back to the main window and clicks on the **Service Items** button, the Service Items window opens.” [See Figure 15 on page 85.]

“This lists all the service items defined, with their associated standard labor cost. When the user clicks on a particular service item, I fetch a list of the parts needed to carry out a service of this type. I display these parts with the quantities required and the cost per part in the lower list box. And that’s all the user does with that window,” concluded Steve, closing it.

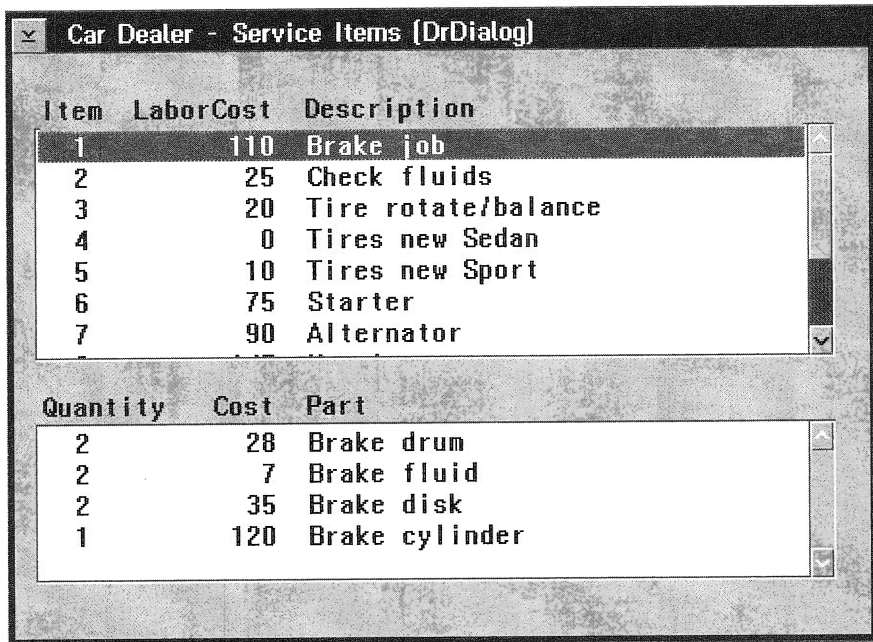


Figure 15. *Service Items List Window of Dr. Dialog GUI Application*

“The last button on the main window [see Figure 13 on page 83] is the work orders **List** button,” he added. “But before we bring that up, notice that I’ve put three radio buttons nearby:

- ☐ “**Customer**, which lists the currently selected customer’s work orders
- ☐ “**Incomplete**, which lists all currently incomplete work orders
- ☐ “**All**, which lists all work orders.”

Steve selected the customer, *Felicia Dolcevit*a, clicked on the **Customer** radio button, and then on the **List** button. “When the user clicks on this, up pops the Work Orders window,” he said [see Figure 16 on page 86].

“This window has space for lots of things. I show the currently selected customer if the user chose the **Customer** radio button; otherwise, the *Customer* field is blank. Below that I show a list of the work orders. If the user clicks on a particular work order in the list box, I put the vehicle make and model in the *Vehicle* entry field, and the work order number in the *Order No* entry field,” Steve continued, clicking as he spoke. “I also show a list of the service items associated with this work order in the *Service Items* list box. Users can retrieve any work order by number, by keying it into the *Order No* entry field and clicking on the **Retrieve** button.”

Customer: Dolcevita, Felicia

Vehicle: Cadillac-Alliante

Order No: 2

Retrieve New Delete

Complete Bill

Order	Date	Cost	Status	Vehicle
2	09/06/95	578	incomplete	Cadillac-Alli

Service Items: Add Item

Item	LaborCost	Description
4	0	Tires new Sedan
9	85	Electrical
10	85	Exhaust system

Figure 16. Work Orders Window of Dr. Dialog GUI Application

“Wow, Steve, you’ve put a lot of work into this,” said Hanna. “It looks impressive.”

“There’s more to come!” said Steve. “If the user has selected a vehicle, he or she can click on the **New** button to create a new work order for that vehicle.” Steve clicked the button, and a new work order appeared in the list box. It was already selected. “There are no service items on the new work order yet. But I can type a service item number into the entry field just right of the **Add Item** button and click on it. See, the service item appears in the list below,” said Steve, pointing to the screen.

“Can you delete service items from an existing work order?” asked Hanna.

“No!” said Steve emphatically. “That’s a feature they particularly *don’t* want to have. They suspect that there’s some ‘sweethearting’ going on in the service department, where services are performed but not

billed. If a service item is added to a work order in error, the clerk responsible has to make out a whole new work order, and get management approval to delete the old one.

“Once all the service items on a work order have been completed, the users have to mark the work order complete,” Steve continued. “They do so by selecting the work order and clicking the **Complete** button, like so. I compute the final cost of the work order, based on the standard charges for the service items and parts involved, and update the work order to show that it’s complete, and what the final cost is.”

“What happens when you click on the **Bill** button?” asked Hanna.

“This,” answered Steve, clicking on the button. A green window opened [see Figure 17]. “This is the bill that I need to print for them. It has the same format as the one Curt produces for Trusty Trucks. I don’t have a printer for my ThinkPad at home, so I’m displaying the print image on screen for the time being.”

Car Dealer - Work Order Bill (DrDialog)				
Bill for work order 2				Date: 09/06/95
Customer: Dolcevita, Felicia				
Vehicle: Cadillac-Alliante				
Description	Parts	Unit	Partcost	Laborcost
Tires new Sedan	4 Tire 185-70	\$ 57 =	228	0
Electrical	1 Cruise control	\$ 54 =	54	85
	3 Light bulb	\$ 2 =	6	
Exhaust system	1 Muffler	\$ 120 =	120	85
Total cost of work order				578

Figure 17. Billing Window of Dr. Dialog GUI Application

“Is that the lot?” asked Hanna.

“I think so, let me just make sure...” answered Steve, opening and closing windows in rapid succession. “Yes, that’s it,” he concluded.

“Well, it looks wonderful to me,” said Hanna. “And it seems pretty robust. I didn’t notice any glitches or crashes.”

“Dr. Dialog has very good debugging facilities,” said Steve. “In addition to that, I really didn’t have to write a lot of code to animate the GUI—only about 300 REXX statements. The final program has about twice as many statements as that, but Dr. Dialog automatically generates a lot of the code you need. Of course, I made use of the class libraries we developed for the Trusty Trucks application. I managed to get a high degree of reuse.”

“That’s exactly what we were trying to achieve, and I’m delighted that it’s working out so well,” said Hanna. “What’s the next step, Steve?”

“I’m due at Classy Cars tomorrow morning to review progress on my GUI development,” said Steve. “I think the code is in good shape. I’m sure they’ll be happy with it. If they are, then I can start developing the new classes required to use DB2 for the persistent storage of our objects.”

“Good luck, Steve,” said Hanna. “We’ve got to win the business at Classy Cars. We’ve invested a lot to make this application configurable, and we’ve got to get a return on that investment.”

Steve smiled. “Don’t worry, Hanna,” he said. “I’m sure they’ll sign. They need our system to get better control of their operation.”

Choice of GUI Builders

But things didn’t go quite as smoothly as Steve had hoped. He came back into the office about noon the next day looking downhearted. Hanna and Curt were working at their desks when he came in.

“Hi Steve, how did your meeting with Classy Cars go this morning?” asked Hanna. “You don’t look too cheerful.”

Steve shook his head as he put his ThinkPad bag down on his desk. “It was mixed,” he said. “They loved my GUI code. They were really enthusiastic about it. They said that it completely transformed their view of the application and how it could help their business. But then the consultant started asking questions...”

“Which consultant, Steve?” asked Hanna.

“Classy Cars engaged a consultant from the Strategic IT Studies firm to review their IT directions,” answered Steve.

“Uh, oh,” chipped in Curt, “we aren’t their favorite developers. Not since we had to point out some of the holes in the design work they did for Hardbright Steel last year.”

“Well, be that as it may, the consultant asked a whole lot of questions,” said Steve. “And one of the things he wanted to know is which GUI development package we were using. I told him all about Dr. Dialog and how good it is, but he kept saying that it isn’t a marketed product, and what kind of support can Classy Cars hope to get for a product for which no one is getting any revenue? He kept asking me what the future plans are for Dr. Dialog, and I didn’t have any answers. I kept on telling him that it’s a great package as it stands right now, and he kept saying, ‘Fine, but what about next year, and the year after that?’”

“Oops! Sounds like you were in a nasty situation,” sympathized Hanna.

"You're right!" agreed Steve. "Anyhow, I reminded them that we had adopted a completely OO approach to our design, maximizing reuse and configurability, and that they shouldn't get hung up on which GUI builder we used. It would be a simple matter to change to another if they ever needed to."

"That's quite true, Steve," agreed Hanna. "We should be able to use other GUI builders without having to change our existing class libraries, and that's where the bulk of our development effort has been invested."

Steve smiled wryly. "I'm glad you agree with me, Hanna," he said, "because they called my bluff. They said that if it's so easy to change to another GUI builder, why don't we go ahead and do it? I said hey, that's extra work we didn't quote for. They said so quote for it, and remember how easy you just told us it would be."

"What a clown!" said Curt. "You really lead with your chin!"

Hanna looked aghast. "Oh no, Steve! They backed you into a corner! We've already invested a lot in this deal. I don't know if we should give it up as a bad job."

"Now you just hold on there a minute!" said Steve fiercely. "We've been sweating blood to design this system so that it's easy to configure and adapt. But the very first time someone takes us up and questions our claims, you all turn chicken and want to run away. Don't you believe in what we did together?"

For long tense moments, Hanna and Curt just stared at Steve in silence. Then Hanna buried her face in her hands, gathered herself, and said, "What you say may be true, Steve. But we've already gone into debt setting up our business. The first few opportunities we've tackled have brought in some money, but it's still touch and go whether we'll survive. Your arguments are completely valid for a well-established company. But we might be out of business before we even start to see the benefits of reuse."

Steve looked glum, then Curt spoke up. "Steve, how long did it take you to build the Dr. Dialog GUI?" he asked.

Steve thought back. "About five working days," he said.

"And how much of that was GUI design and layout, as opposed to building and debugging logic?" Curt continued.

Steve thought more deeply. "There's always a lot of fiddling around, trying to work out what the user needs to see and how to present it on the screen," he said. "I guess it was about half and half."

"So if we had to port the front end to another GUI builder, how long would it take?" asked Curt.

"If we keep the layout and logic the same, about two days," Steve answered. "I'm pretty sure we could cut and paste most of the current logic from Dr. Dialog into the new GUI builder, then customize it to use the new builder's calling conventions."

"And if we do it, will we get the business?" asked Curt.

Steve hesitated. "There's still all the DB2 work to do," he said. "Look, after all I've said to them, if we *don't* do the GUI port, and do it fast, we *won't* get the business. And that's for sure."

Curt turned to Hanna. "Tomorrow's the weekend," he said. "Maybe we can do it by Monday."

Hanna looked at him uncertainly. "But what GUI builder would we use, if not Dr. Dialog?" she asked. "It's very short notice to research the GUI builders available, choose one, get it, install it, and learn to use it before tomorrow."

"The situation isn't that bad," Curt answered. "When we first went out looking for GUI builders for REXX, I found several that could potentially meet the bill. We asked the vendors for evaluation copies. Steve and I both looked at them briefly before we discovered that Dr. Dialog is on DEVCON [the Developer Connection for OS/2] and settled for that."

"That's right," agreed Steve. "I looked at VisPro/REXX pretty closely, and I liked what I saw."

"And I looked at Watcom VX•REXX," said Curt. "I'm sure that it could do the job."

Hanna managed to summon a smile. "So which one should we use?" she asked.

"VisPro/REXX," said Steve.

"Watcom VX•REXX," said Curt, simultaneously.

"Oh great! Let's use them both!" said Hanna with unaccustomed sarcasm.

"I'm going to do it in VisPro/REXX," said Steve. "I'm sure it will do the job."

"You didn't make a detailed evaluation of it, Steve," said Hanna. "What if you're doing things in Dr. Dialog that you just can't do in VisPro/REXX? You've already shown Classy Cars what you've accomplished. If it happens that you can't get the same things working in VisPro/REXX early next week, will they give us another chance? I think it's just too risky. Maybe we should just call it off."

"Well I looked at Watcom VX•REXX for more than a day" said Curt. "It had all the features needed to build Common User Access [CUA] applications. I'm sure that it can do what Classy Cars needs."

"Talking about it isn't going to help," said Steve. "I'll do it in VisPro/REXX this weekend, trust me."

“Oh, it’s ‘Trust me, I’m a programmer’ time, is it?” sneered Curt. “That doesn’t fill me with confidence! I’m going to do it in Watcom VX•REXX this weekend.”

“Don’t be absurd, Curt!” snapped Steve. “If you really have a weekend to waste, why don’t you come in to the office and work on this thing together with me?”

“And what if Monday comes round, and all we’ve done is prove that it can’t be ported to VisPro/REXX for some reason?” asked Curt.

There was a tense silence. Hanna could feel a giggle coming on, and rather than earn their wrath, she said “I guess if a thing’s worth doing, it’s worth doing twice.” Curt and Steve smiled thinly at her jest, but neither was going to back down.

“Where’s the code?” asked Curt.

“On the server,” Steve answered.

Both settled down to work in angry silence. Hanna thought seriously about praying.

On Monday morning, Hanna was in early. She hadn’t heard from either Steve or Curt over the weekend, and she waited nervously to discover what they had achieved. There was a clatter at the door, and they walked in together, thumping their bags down onto their desks. They plugged in their ThinkPads. While they were powering up, Curt said “I got the application working with Watcom VX•REXX.”

“Well, I got it working with VisPro/REXX,” replied Steve.

“Well done, guys!” said Hanna. “Sounds like it’s time for a shootout. Let’s compare them to the original Dr. Dialog.”

The threesome lined up their ThinkPads in a row and each brought up a different interface. Hanna put the Dr. Dialog interface through its paces, and Curt and Steve on either side of her mirrored her actions in the new GUIs they had built. Except for a few cosmetic differences and a couple of little bugs, all three GUIs looked and behaved the same.

“Well, that proves we can do the job with any GUI,” said Hanna. “I don’t suppose we can bill them for all three...”

Steve shook his head ruefully. “Well, if this doesn’t knock their socks off, I don’t know what will,” he said. “I’ll take all three interfaces out and show them to Classy Cars. Oh. And thanks, Curt. Your GUI looks great.”

Steve copied the GUI directories across the LAN and headed out for Classy Cars. He was gone half the day. When he came back, he was smiling broadly.

“They were delighted,” he said. “I set up our three different flavors on three of their PCs standing all in a row. Then I walked them through the same exercise we did here this morning. Everything worked per-

fectly. Well—near enough perfectly. I explained that there’s still quite a bit of rounding off to do, but we need their signature on an order before we can invest the time that will be required. They understood.”

“So which one did they choose?” asked Curt impatiently.

“Huh? Oh, they’re really convinced now that we were right when we said that we could port the application to another GUI any time if we had to,” Steve replied. “So they decided to go with the cheapest solution—Dr. Dialog!”

“What! After all that effort?” said Hanna, and the three of them roared with laughter which had a lot of relief mixed with it, too.

How to Include Directives in GUI Builders

The next day, while all three were gathered in their office again, Steve said, “You know, the only really tricky part of that GUI coding exercise was getting my Object REXX directives to the end of the REXX code generated by VisPro/REXX. Object REXX insists that all directives must come at the *end* of the source file that contains them. But VisPro/REXX generates the final REXX source itself. It includes all the procedures I coded, but I can’t tell it their sequence.”

“Yes, I had exactly the same problem with Watcom VX•REXX,” Curt agreed.

“That’s interesting,” said Hanna. “But Steve, you must have hit this problem when you did the original GUI in Dr. Dialog.”

Directives in Dr. Dialog

I certainly did,” said Steve. “It took me a while to work out what was going wrong, and then I read the manual. Dr. Dialog stores all the GUI and logic that you feed into it in a .RES file. Each time you save or run the project .RES file, Dr. Dialog checks to see if you have included a file with the same name as the .RES file but with extension .REX. If it finds it, Dr. Dialog copies its contents into the back of the .RES file as a special resource type. When the application runs, this special resource is at the end of the REXX program Dr. Dialog creates. So that was really quite straightforward. You put the Object REXX directives into the .REX file. This becomes part of the .RES file when you save the project, so you don’t have to distribute the .REX file when you distribute the application to other users.”

Directives in Dr. Dialog

1. Assume the GUI dialog is stored under the name `car-gui.res`.
2. Create a file named `car-gui.rex` and put all the directives into that file:

```
::requires customer.cls
::requires vehicle.cls
::requires part.cls
...
```

3. Keep the class definitions in separate files.

Directives in VisPro/REXX

“That sounds nice and easy,” said Hanna. “How about VisPro/REXX?”

“VisPro/REXX stores the GUI project in several subdirectories,” Steve answered. “After looking around I found out that VisPro/REXX always looks to see if you have put any REXX code in the project’s SubProcs subdirectory. If you have, VisPro/REXX includes it at the end of the REXX file that it generates when the project is run, or when an .EXE module is created. There are no rules about the name of the file. Any file name and extension can be used, but be careful, a return statement is automatically added at the end of the file.”

Directives in VisPro/REXX

1. VisPro/REXX generates the executable code in alphabetical sequence of the subdirectories.
 - Each GUI window is a separate subdirectory; the initial window and directory is named `Main` by default.
 - There is a subdirectory for procedures, `SubProcs`.
2. Therefore, make sure that all of the windows are stored in subdirectories with names that are alphabetically before `SubProcs`.
3. Put all directives into the alphabetically last file in the `SubProcs` directory.
4. VisPro/REXX adds a *return* statement at the end of the file. If that does not fit, put a dummy method last.
5. Keep the class definitions in separate files.

For the car dealer application the directives are in the `zCargui.cvp` file in the `SubProcs` directory:

```
::requires customer.cls
::requires vehicle.cls
::requires part.cls
...
::class dummy
::method dummy /* VisPro/REXX adds a return statement */
```

Directives in Watcom VX•REXX

“And what did you have to do with Watcom VX•REXX, Curt?” asked Hanna.

“It wasn’t too bad, once I broke down and read the manual,” Curt replied. “Watcom VX•REXX allows you to include external code in a Watcom VX•REXX project using something called *shared sections*. You put your code into the shared section, then tell Watcom VX•REXX to include it in the project. I’ll show you how to add a section that contains directives.”

Curt started Watcom VX•REXX project on his ThinkPad and opened the Sections window using the *Section List* action in the *Windows* pull-down of the main project window. He selected the *Add...* action from the *Section* pull-down. A prompt box popped up, and Curt entered *car-gui.cvx*. “That’s the name of the file where I coded my directives,” he said. “The shared section will be added to the front of the section list.” He showed them his screen:

```
<car-gui.cvx>                <===== file with directives
addcar_Click
addcust_Click
additem_Click
CarDealBill_Close
...
workorders_Click
```

“When the project is run or an executable or macro command file is made,” Curt continued, “The contents of the directives are added to the end of the file. The last statement must be a return, however.”

Directives in Watcom VX•REXX

1. Watcom VX•REXX generates the executable code in alphabetical order of sections.
2. Added sections, which appear within less-than and greater-than signs <name.ext>, are appended at the end of the code. Each file must have a return statement at the end.
3. Therefore, put all directives into a separate file and add it to the section list.
4. Keep the class definitions in separate files.

For the car dealer application the directives are in *car-gui.cvx*:

```
::requires customer.cls
::requires vehicle.cls
::requires part.cls
...
::class dummy
::method dummy
return /* required at the end */
```

“Great!” said Hanna. “So it’s pretty easy to do for each of the GUI builders, once you know how.”

GUI Builder Development Environment

In this section we present some snapshots of the GUI development environments for Dr. Dialog, VisPro/REXX, and Watcom VX•REXX.

Development Environment for Dr. Dialog

The directory for a Dr. Dialog project is shown in Figure 18.

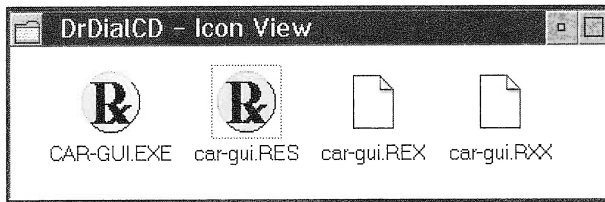


Figure 18. *Dr. Dialog Project Folder*

The development environment is accessed through the *DrDialog* action in the pop-up menu of the .RES file, as shown in Figure 19.

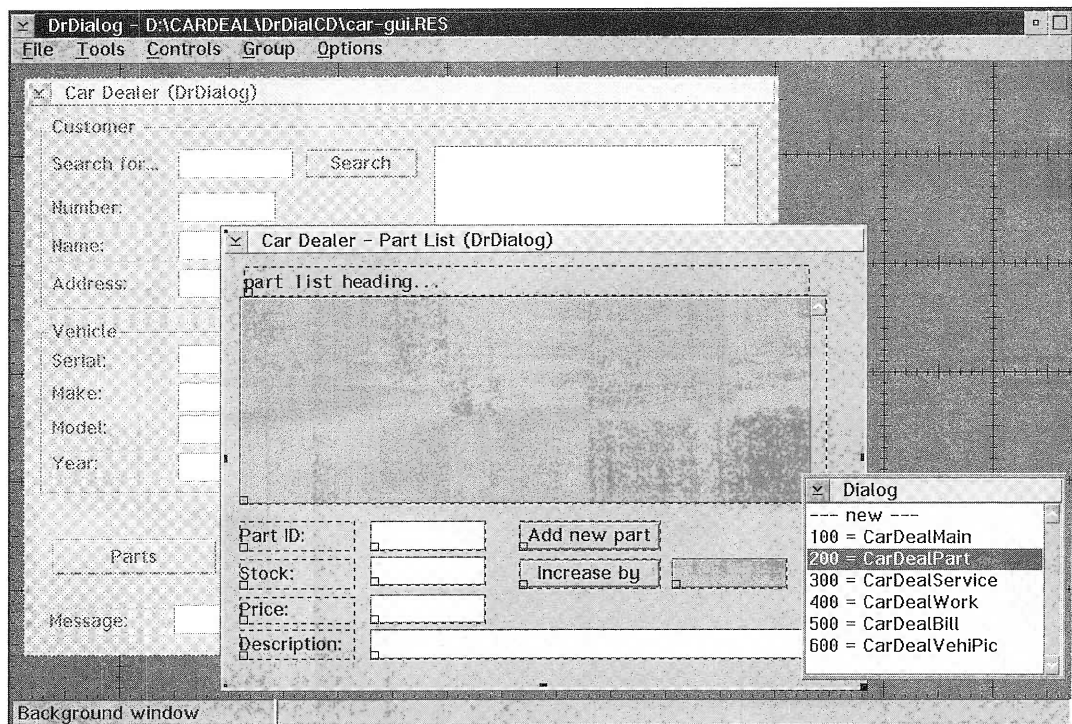


Figure 19. *Dr. Dialog Development Environment: Window Layout*

All application windows are accessible through the Dialog window. All REXX code is accessible through the DrRexx notebook, as shown in Figure 20.

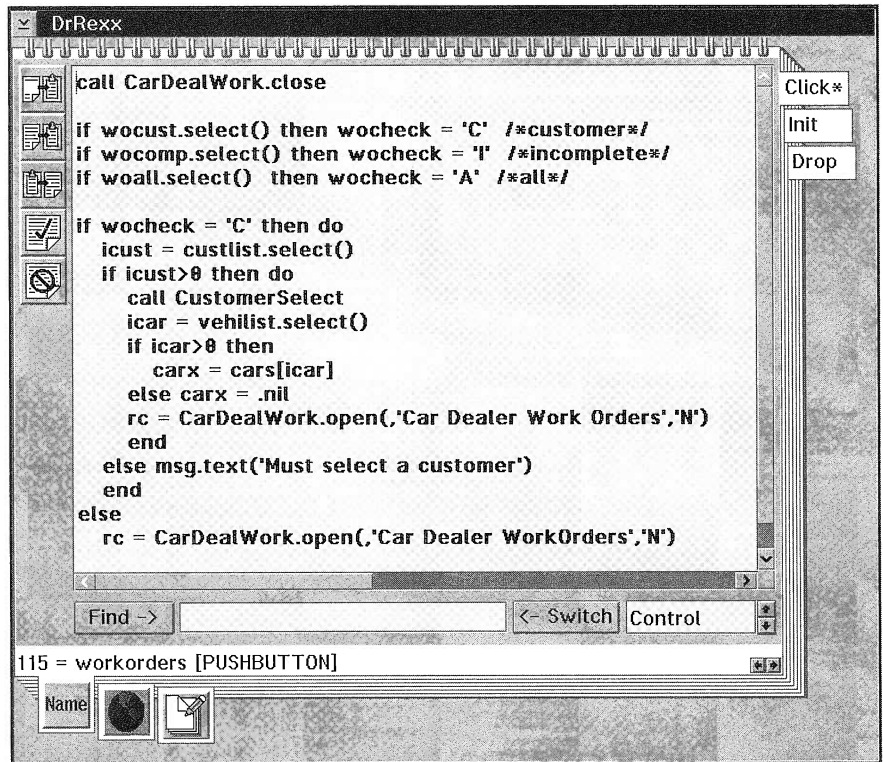


Figure 20. *Dr. Dialog Development Environment: DrRexx Notebook*

Any control in an application window can be double-clicked to display the appropriate page of the DrRexx notebook. Alternatively, all controls can be accessed through the **Name** icon at the bottom of the notebook, and all procedures through the **Globe** icon.

Development Environment for VisPro/REXX

The directory for a VisPro/REXX project is shown in Figure 21.

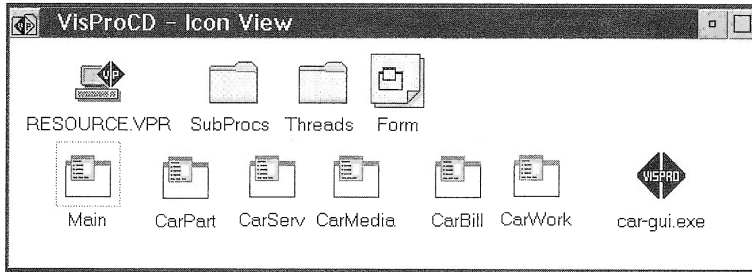


Figure 21. *VisPro/REXX Project Folder*

Each window is a separate icon in that folder, and the SubProcs folder holds all additional procedures.

Double-click on a window folder to start the Layout View, as shown in Figure 22.

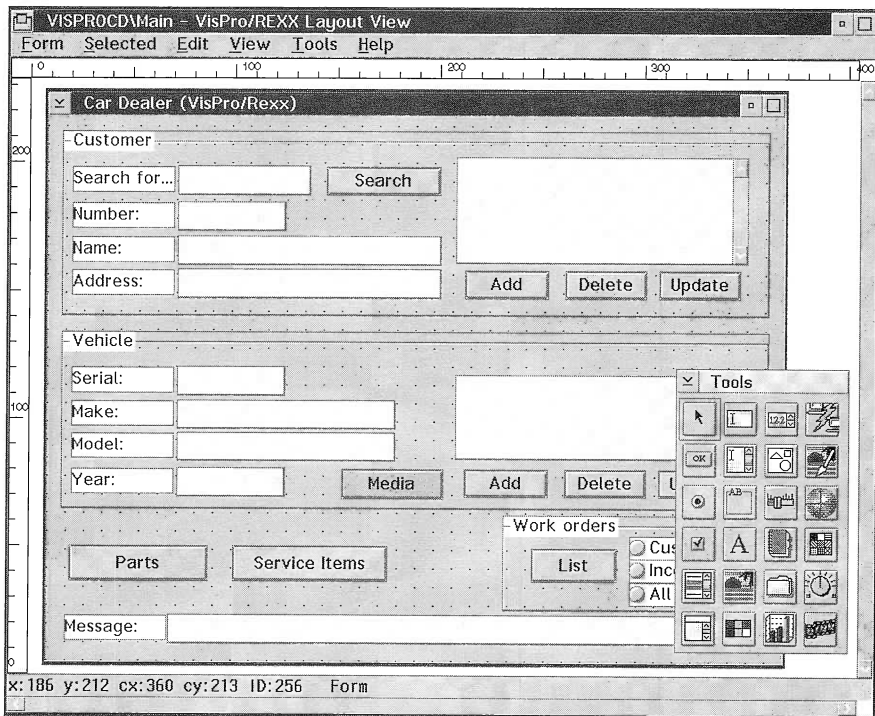


Figure 22. *VisPro/REXX Development Environment: Layout View*

To access the REXX code, open the Event Tree View of the window, which is shown in Figure 23.

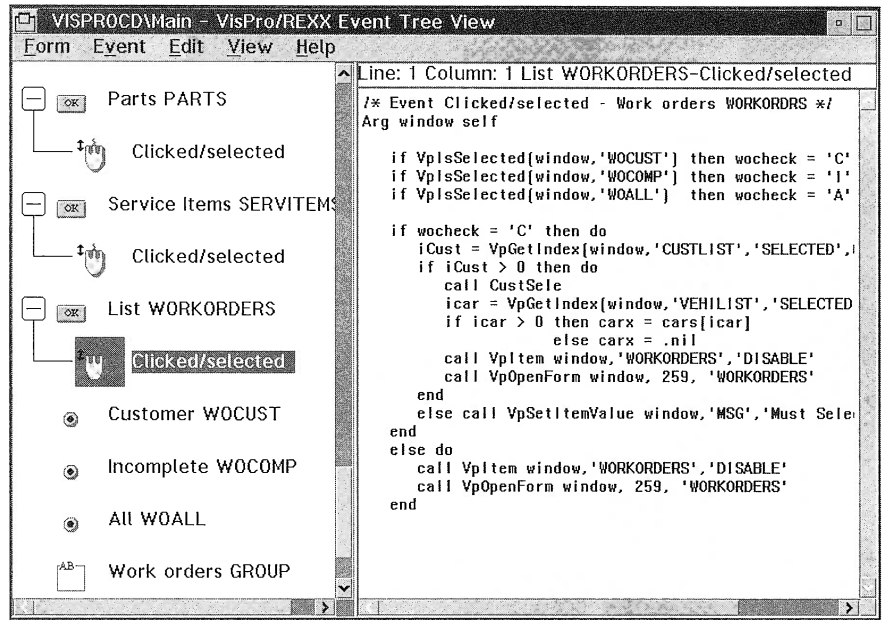


Figure 23. VisPro/REXX Development Environment: Event Tree View

Development Environment for Watcom VX•REXX

The directory for a Watcom VX•REXX project is shown in Figure 24.

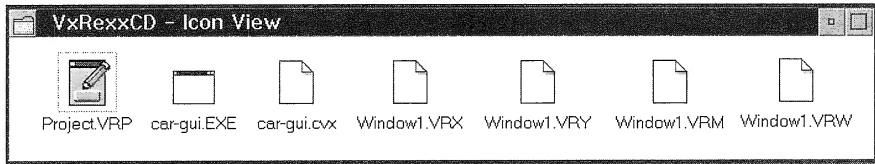


Figure 24. *Watcom VX•REXX Project Folder*

All definitions are stored in a file named `Project.VRP`. Open this file to access any window (Figure 25).

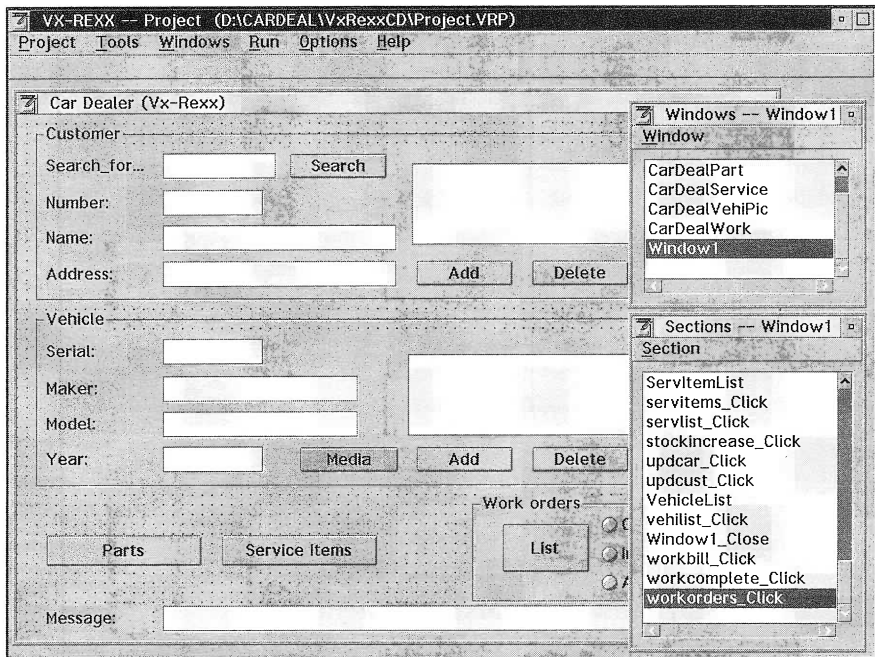


Figure 25. *Watcom VX•REXX Development Environment: Window Layout*

The windows of the application are accessible through the *Windows* pull-down of the main project window.

The REXX code for selected events is defined by using the pop-up menu of each control. The event is then automatically added to the Sections window and can be edited by double-clicking on the item, as shown in Figure 26.

```

workorders_Click
Line: 3      Col: 1      Insert

/*: VRX */
workorders_Click:
|
|   if VRGet( "wocust", "Set" ) then wocheck = 'C'
|   if VRGet( "wocomp", "Set" ) then wocheck = 'I'
|   if VRGet( "woall" , "Set" ) then wocheck = 'A'
|
|   if wocheck = 'C' then do
|       icust = VRGet( "custlist", "Selected" )
|       if icust > 0 Then do
|           call CustomerSelect
|           icar = VRGet( "vehilist", "Selected" )
|           if icar > 0 then carx = cars[icar]
|               else carx = .nil
|           call VRSet "workorders", "Enabled", 0
|           call VRLoadSecondary "CarDealWork"
|       end
|       else call VRSet "msg", "Value", "Must Select a customer"
|   end
|   else do
|       call VRSet "workorders", "Enabled", 0
|       call VRLoadSecondary "CarDealWork"
|   end
|   return

```

Figure 26. Watcom VX•REXX Development Environment: Event Code

Testing and Generating the GUI Applications

All three visual builders enable the user to test the application from the development environment and to generate an executable module (.EXE file):

- ☐ Dr. Dialog generates an .EXE with the same name as the .RES file.
- ☐ VisPro/REXX generates a file named RUN.EXE, which can then be renamed.
- ☐ Watcom VX•REXX prompts for the file name of the executable (default name Project.EXE). You must generate a macro file (default name Window1.VRM) and an executable, both in the directory where the Project.VRP file resides.

7

Persistent Objects in DB2

In this chapter, we find out how objects can be made persistent by storing them in a DB2 database. We use DB2/2 Version 2 for this exercise. Because DB2/2 is part of the DB2 family and provides connectivity to all other members of the family through DDCS/2, the approach described in this chapter could be used, regardless of the platform on which the DB2 databases are stored. We also restricted the SQL functions used in this chapter to a simple subset of the ANSI SQL standard. Therefore, the code should be portable with more or less effort across any of several other vendors' relational DBMSs. In Chapter 8, *Using Advanced DB2/2 Facilities*, on page 111, we exploit some of the more advanced functions of DB2/2 Version 2.

Storing Objects in DB2

"Hi Steve," said Hanna as she walked into the office. "I've just been over to see Trusty Trucks. Our car dealer application is running so smoothly, they're just delighted!"

“That’s great,” replied Steve. “We spent a lot of time designing that system—it *should* run smoothly! But the real benefits of our approach will surface only when we start building and delivering different versions to meet different customers’ needs.”

“Right,” agreed Hanna. “Speaking of which, how are you doing with the DB2 work for Classy Cars?”

“It’s been really easy to do, Hanna,” replied Steve. “All the trouble we took up front to make sure that we could fit DB2 support into the system later has paid off. Look, here’s a picture of the class inheritance I need to build the DB2 support.” Steve showed Hanna Figure 27.

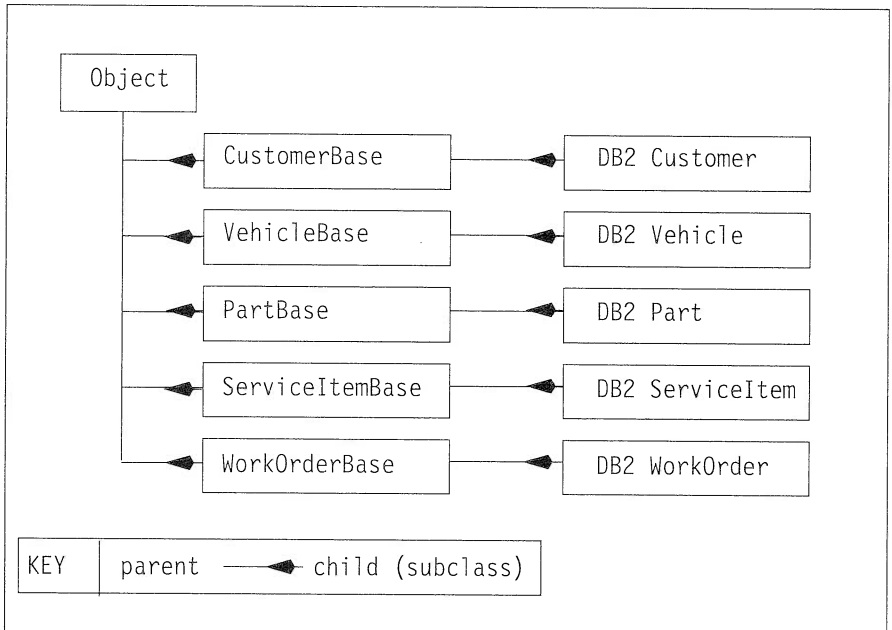


Figure 27. *DB2 Class Inheritance Diagram*

The DB2 classes could also inherit from the Persistent class. This does not provide an advantage, however, because all methods have to be coded in the DB2 class, regardless.

“The base classes contain the methods that are common across DB2 and FAT files, and the DB2 classes contain the DB2-specific methods,” Steve explained. “Since the DB2 classes are subclassed from the base classes, they inherit all the methods of the base classes.”

“That sounds quite straightforward,” said Hanna. “What else do you need to do?”

“Well,” said Steve, “I took the data definitions that Curt drew up when we first went through the car dealer requirements.” [See Figure 7 on page 42.] “All I had to do was turn his COBOL into DB2 SQL. Oh, and get rid of the repeating groups in the service and work order objects.”

“What have you done with them?” asked Hanna.

"I've made them separate tables," said Steve. "Look, here's the table diagram I have drawn up." Steve opened a Freelance picture on his ThinkPad [see Figure 28].

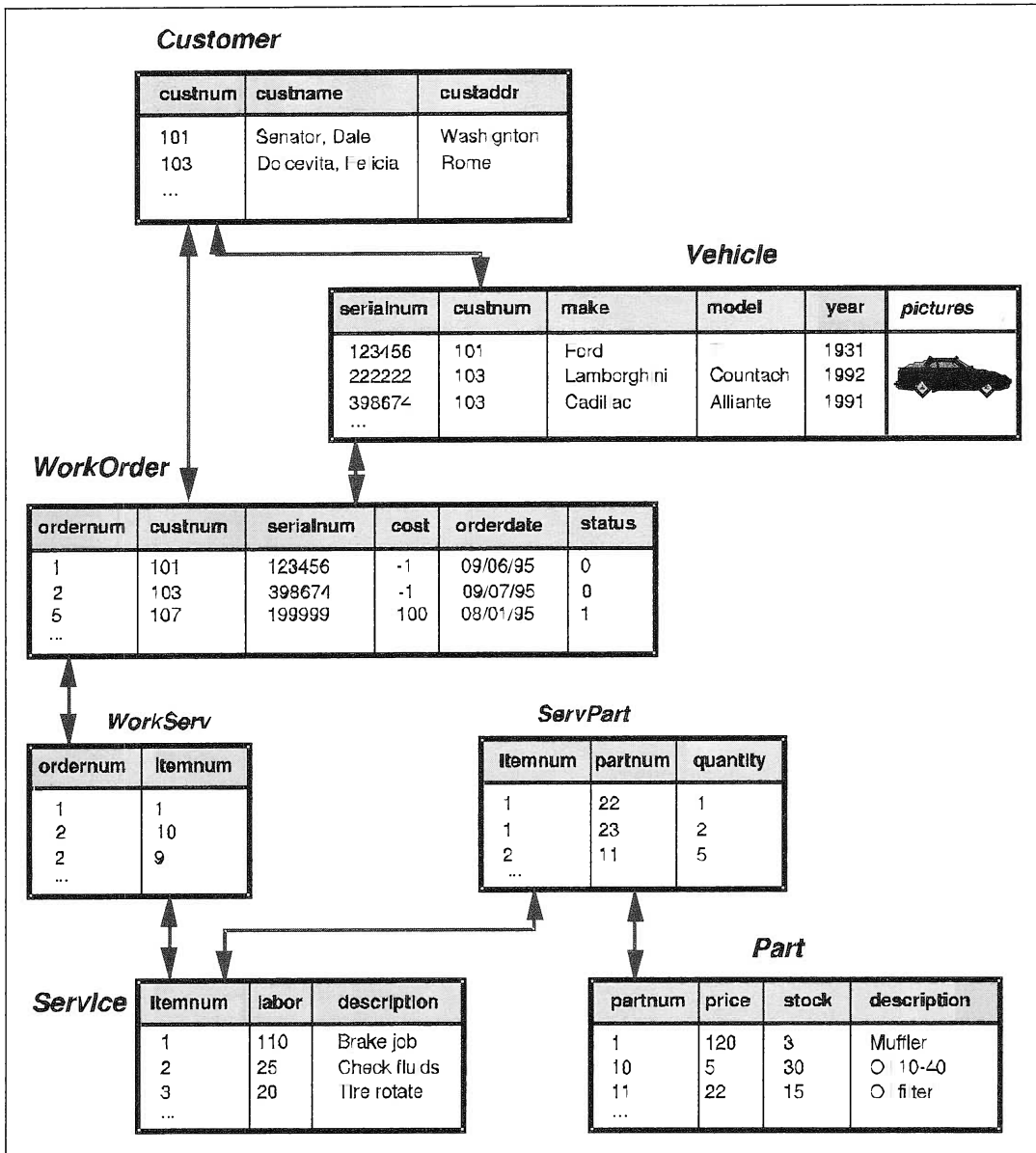


Figure 28. DB2 Tables for Car Dealer Application.

The *pictures* column in the vehicle table is discussed in Chapter 8, *Using Advanced DB2/2 Facilities*, on page 111.

“We need a DB2 table for each class,” explained Steve. “I’ve given them the same names as the classes themselves. And then there are two extras to hold the repeating groups. I’ve called them *Servpart* and *Workserv*. *Servpart* will be used to store the relationship between the service objects and all the parts that each one needs. *Workserv* will be used to store the relationship between the work orders and the services that each one specifies.”

“And here are the SQL commands I think I’ll need.” Steve opened an editor window on his ThinkPad [Figure 29].

```
DROP TABLE CARDEAL.CUSTOMER
DROP TABLE CARDEAL.PART;
...
CREATE TABLE CARDEAL.CUSTOMER
(CUSTNUM          SMALLINT NOT NULL,
 CUSTNAME         CHAR(20) NOT NULL,
 CUSTADDR         CHAR(20) NOT NULL) ;
CREATE TABLE CARDEAL.PART
(PARTNUM          SMALLINT NOT NULL,
 PRICE           SMALLINT NOT NULL,
 STOCK           SMALLINT NOT NULL,
 DESCRIPTION      CHAR(15) NOT NULL) ;
...
CREATE UNIQUE INDEX CUSTOMER_IX ON CARDEAL.CUSTOMER (CUSTNUM);
CREATE UNIQUE INDEX PART_IX ON CARDEAL.PART (PARTNUM)
...
```

Figure 29. *DB2 Table Definitions*

Extract of SQL DDL statements for table definitions.

Hanna studied the SQL commands. “Have you set up a database for this yet?” she asked.

“Sure,” answered Steve [Figure 30].

```
CREATE DATABASE DEALERDB ON D          -- D is the disk drive letter
```

Figure 30. *DB2 Database Definition*

“And I’ve run the SQL. I wrote a little REXX command file called `run-sql.cmd` [see Table 35 on page 259] to read this file and pass it over to the DB2 command line utility, and I’ve already run these table definitions through it. A couple of times!” he added. “That’s why I’ve got the *DROP TABLE* commands at the top. There were a few errors in my SQL the first time around.”

Hanna smiled. “I believe you!” she sympathized.

“I’ve also coded up the SQL required to insert our test values into the DB2 tables,” said Steve, dragging an icon from a folder and dropping it on the editor. A long string of insert commands appeared, and Steve scrolled down through them [Figure 31].


```

delete * from cardeal.customer
...
commit
insert into cardeal.customer (custnum, custname, custaddr)
      values (101, 'Senator, Dale', 'Washington')
...
insert into cardeal.vehicle (serialnum, custnum, make, model, year)
      values (123456, 101, 'Ford', 'T', 1931)
...
insert into cardeal.workorder (ordernum, custnum, serialnum, cost, orderdate, status)
      values (1, 101, 123456, -1, '09/06/95', 0)
...
insert into cardeal.service (itemnum, labor, description)
      values (1, 110, 'Brake job')
...
insert into cardeal.part (partnum, price, stock, description)
      values (21, 120, 3, 'Brake cylinder')
...
insert into cardeal.workserv (ordernum, itemnum)
      values (1, 1)
...
insert into cardeal.servpart (itemnum, partnum, quantity)
      values (1, 21, 1)
...
commit

```

Figure 31. *DB2 Sample Table Load*

Extract of SQL statements to load sample DB2 tables.

“And this one starts with a whole bunch of delete commands—just in case?” asked Hanna.

“Right!” agreed Steve. “I’ve already made sure that they work, too. But the test data is loaded, and now I’m working on the definitions of the DB2 classes. We’re going to need a whole lot of new methods.”

“Oh dear!” said Hanna apprehensively. “I hope this doesn’t turn out to be a lot of extra work.”

Steve’s frustrations boiled over. “Hanna, you and Curt keep challenging me about the DB2 support. But Classy Cars is a much bigger operation than Trusty Trucks. Their turnover was five times bigger last year. They’ve got branches in twenty cities around the country. Sure, it’s going to take work to adapt our application to meet their needs. But we’ll get far more revenue out of them than we’ll ever see from Trusty Trucks. And once we’ve adapted our application to fully support a GUI front end and a real database, it will be a far more marketable product than it is today. How many businesses want a clunky character interface when they buy a computer package nowadays?”

“You’re 100% right, Steve,” she said soothingly. “All of us recognize that Classy Cars is a wonderful business opportunity. But we’re a very small operation. I’m worried that we may go bankrupt before we get a chance to show them how good we are. We have to make absolutely sure that the Trusty Trucks implementation completes on the due date with no hitches, so we can get paid on schedule.”

"I know that, Hanna," Steve replied. "The best guarantee for a smooth installation is a good design. That's why I keep on insisting that we get the design right, instead of jumping into coding."

Hanna smiled. "You're right, Steve," she said. "We've all been working hard on this project to make sure it's a success. So let's settle down and do some more designing! Have you decided whether you're going to load all the objects from DB2 into storage when the system starts up, or fetch them as you need them?"

Steve relaxed as he turned back to his design. "The people at Classy Cars haven't yet decided whether they want one centralized database or whether each operation will get its own," he mused. "If it's centralized, the volumes will be pretty big, and we'll have to go for a *load-on-demand* approach. But if they decentralize, no single operation is so big that its objects wouldn't fit into storage."

"So what's the answer?" asked Hanna.

Persistent Methods for DB2 Support

"We can't wait for them to make up their minds," Steve answered. "We have to assume the worst case, and make sure we can handle it."

That means loading objects only when required and updating them directly on disk every time they change. Of course, there's only a limited number of part- or service-type objects, no matter how big the operation is. We can carry on loading all those into storage when the application comes up. But customers, vehicles, and work orders will have to stay out on disk."

"Will this mean a lot of extra coding, Steve?" asked Hanna.

"I've worked out that we'll need the following methods," replied Steve, as he showed her Tables 13–17.

Table 13. Methods for Customer Persistent Storage in DB2

Method	Type	Purpose
findName	Class	Find list of customers in DB2 given an abbreviated name
findNumber	Class	Find customer in DB2 given the number; create customer object in memory with cars and work orders
persistentInsert	Instance	Insert a new customer into DB2
persistentUpdate	Instance	Update an old customer in DB2
persistentDelete	Instance	Delete an old customer from DB2
ListCustomerShort	Class	List customers on standard output
ListCustomerLong	Class	List customers and cars on standard output

Table 14. Methods for Part Persistent Storage in DB2		
Method	Type	Purpose
persistentLoad	Class	Load all parts from DB2
persistentInsert	Instance	Insert a new part into DB2
persistentUpdate	Instance	Update an old part in DB2

Table 15. Methods for Service Item Persistent Storage in DB2		
Method	Type	Purpose
persistentLoad	Class	Load all service items from DB2

Table 16. Methods for Vehicle Persistent Storage in DB2		
Method	Type	Purpose
persistentLoadByCust	Class	Load all vehicles of a customer into memory
persistentInsert	Instance	Insert a new vehicle into DB2
persistentUpdate	Instance	Update an old vehicle in DB2
persistentDelete	Instance	Delete an old vehicle from DB2

Table 17. Methods for Work Order Persistent Storage in DB2		
Method	Type	Purpose
persistentLoadByCust	Class	Load all work orders of a customer into memory
findNumber	Class	Get work order by number
findStatus	Class	Get all work orders by status
newNumber	Class	Make new work order number
persistentInsert	Instance	Insert a new work order into DB2
persistentUpdate	Instance	Update an old work order in DB2
persistentDelete	Instance	Delete an old work order from DB2
persistentInsertServ	Instance	Add a new service to the work order
persistentDeleteServ	Instance	Remove a service from the work order
ListWorkOrder	Class	List work orders on standard output

“Wow, Steve—that looks like a lot!” said Hanna concerned.

“I’ve already coded up some of the simpler methods,” Steve replied, “and I estimate that the whole job will take about twice as many lines of code as the methods we developed to support persistent storage in ASCII files. That’s not bad, when you consider all the extra things that DB2 will give us:

- ❑ “Support for multiple workstations performing updates concurrently
- ❑ “Automatic rollback of programs that fail
- ❑ “Logging of all updates
- ❑ “Recovery of corrupt databases from the log
- ❑ “The ability to handle large volumes of data
- ❑ “The ability to run the database on servers as big as an ES/9000.”

“Enough, already!” Hanna interrupted him. “You don’t have to sell me on the advantages of DB2, you know that I love using it. Are you including all the SQL we’ll have to code in your estimates?”

“For sure,” responded Steve.

“OK, Steve,” said Hanna. “I’ve got to get back to Trusty Trucks. Curt has everything there pretty well under control, but I want to make sure one more time that the users are ready for the system. If we need you, I’ll call you, but in the meantime it would be fine for you to carry on with the DB2 design. It will get us well ahead of the schedule we agreed to with Classy Cars. There’s nothing better than getting off to a flying start.”

Steve smiled his appreciation. “I’ll be here if you need me,” he said. “Good luck with the users!”

Implementation of DB2 Support

The steps to add DB2 support are:

1. Define the DB2 database.
2. Define the tables in the database; an extract is listed in Figure 29 on page 104.
3. Load the tables with sample data; an extract of possible SQL commands is listed in Figure 31 on page 105.

These three steps are part of the installation program. Then:

4. Write the Object REXX code for DB2 persistence:
 - No changes are necessary to the base classes. They already have the coding to invoke the persistent methods from the FAT implementation. For example, the *init* method invokes *persistentInsert* for new application objects
 - Prepare the classes as subclasses of the base classes:

```
::class Customer public subclass CustomerBase
```
 - Write all the additional methods for DB2 persistence
 - Implement the creation of memory objects at application start for parts and services, and load-on-demand for customers, vehicles, and work orders

Implementation of Load at Application Start

During initialization of the application, all parts and services are loaded into memory by using the *persistentLoad* methods, similar to the flat file support but with data from the DB2 database.

Loading of parts at application start (abbreviated)

```

::class Part
::method persistentLoad class
  stmt = 'select p.partnum, p.price, p.stock, p.description'
        '      from cardeal.part p order by 1' /* SQL Select */
  call sqlxec 'PREPARE s1 FROM :stmt'
  call sqlxec 'DECLARE c1 CURSOR FOR s1'
  call sqlxec 'OPEN c1'
  do ipart = 0 by 1 until sqlca.sqlcode \= 0
    call sqlxec 'FETCH c1 INTO :xpartid, :xprice, :xstock, :xdesc2'
    if sqlca.sqlcode = 0 then
      partx = self~new(xpartid, xdesc2, xprice, xstock) /* part object */
    end
  call sqlxec 'CLOSE c1'
  return ipart

```

Implementation of Load-on-Demand

Customers, vehicles, and work orders are loaded on demand, based on the assumption that there would be too many for all of them to be loaded into memory.

To leave intact the pointer implementation of the base classes between customers, their vehicles and work orders, and the services of a work order, we always load all the data associated with a customer.

Customers are loaded into memory by their number. The *findNumber* method implements the DB2 load of a customer, then invokes the *Vehicle* and *Work Order* classes to load all the data associated with that customer.

Loading of customers on demand (abbreviated)

```

::class Customer
::method findNumber class
  use arg custnum /* input is customer number */
  stmt = 'select c.custname, c.custaddr', /* SQL select statement */
        '      from cardeal.customer c where c.custnum = ' custnum
  call sqlxec 'PREPARE s1 FROM :stmt' /* prepare the SQL */
  call sqlxec 'DECLARE c1 CURSOR FOR s1' /* define and open a cursor */
  call sqlxec 'OPEN c1'
  call sqlxec 'FETCH c1 INTO :xcustn, :xcusta' /* fetch the matching row */
  if sqlca.sqlcode = 0 then do /* found a customer */
    custx = self~new(custnum, xcustn, xcusta) /* make an OREXX object */
    .Vehicle~persistentLoadByCust(custx) /* load the vehicles ... and */
    .WorkOrder~persistentLoadByCust(custx) /* work orders of the cust. */
  end
  else custx = .nil /* customer not found */
  call sqlxec 'CLOSE c1' /* close the cursor */
  return custx /* return the customer obj. */

```

When accessing work orders directly by number, we retrieve the customer number of the work order from DB2, then all the data of that customer is loaded as shown above, including the requested work order.

Implementation Notes

1. To define the tables and indexes, we wrote the `runsql.cmd` program, which reads a file with SQL DDL statements and submits them to the DB2 command processor (`DBM.CMD` for DB2 version 1, `DB2.EXE` for DB2 version 2).
2. For the sample application, the DB2 tables are loaded from the same files used for the flat file persistent storage. The installation program, `load-db2.cmd`, reads the files and loads the DB2 tables.
3. To prepare and set up the DB2 system, we wrote the `db2setup.cmd` that invokes the `runsql.cmd` with the proper DDL files to define the tables and indexes, and then the `load-db2.cmd` to load the sample data into the tables.
4. We did not use DB2 referential integrity to check the relationships between primary and foreign keys in the tables.
5. Customers can also be retrieved by partial name. DB2's *LIKE* facility is used to search the database, and an array of matching customer names, together with their number, is returned. Data is loaded into memory with the *findNumber* method only when a customer from the resulting array is selected.
6. All updates to the data are performed first in memory and then immediately thereafter in DB2 with the *persistentInsert*, *persistentupdate*, and *persistentDelete* methods. The DB2 database is, therefore, always up to date.

Source Code for DB2 Class Implementation

The source code for the DB2 classes is described in Table 23 on page 256 and listed in *Persistence in DB2* on page 317.

The source code for the table definitions and the DB2 setup and load programs is not listed in the appendix; it is available in the car dealer directory on the CD or after the sample applications have been installed on a hard drive (see Table 35 on page 259).

8

Using Advanced DB2/2 Facilities

In this chapter, we exploit some of the more advanced features of DB2/2 delivered in Version 2 of the product. We make use of DB2's BLOBs to store multimedia data.

Multimedia in DB2 BLOBs

"I hope things go well today when you call on Classy Cars, Steve!" called out Hanna. "We need you to bring back a signed contract."

"I'll do my best," replied Steve. "Classy Cars is really keen on our car dealer package. I'm just a bit worried about delivering the multimedia function that we've promised to give them. Time's getting short."

"Is multimedia really necessary, Steve?" Hanna asked. "Wouldn't it be simpler to install the application without it, and then come back to it later if they really want it?"

"They *do* really want it, Hanna," Steve replied. "As I've mentioned, they make more money from selling cars than from servicing them. They want to boost their sales, and they believe that the multimedia facilities I described and demoed to them will be a big help. Where

they really hope to score is by exchanging information between different branches about cars they have for sale. So if a customer expresses interest in some type of car that the branch doesn't have, they can quickly search the records of cars for sale at the other branches. If they find the car, they can use multimedia to show it to the customer. If the customer likes it, they will arrange to transfer the car to the most convenient branch for the customer. Classy cars is convinced that their sales will skyrocket."

"That sounds very ingenious," said Hanna. "But how are they going to capture multimedia images of the cars they have in stock? I know that you can take color pictures and have a print shop scan it and turn it into an image file, but that's slow and quite expensive."

Steve grinned. "Ah! I haven't shown you my latest toy," he said. He zipped open his bag and pulled out a black object that looked a bit like a camera. "Here we have a camera that captures image files directly. The lens focuses the image onto a charge-coupled device array [CCD] instead of conventional film, and the camera copies that into its own RAM storage in compressed format. It can hold up to 48 high-quality images. The camera comes with a cable to plug it into a PC's serial port and software to download the images."

"Wow!" said Curt. "I bet that costs plenty."

"About the same as a conventional camera," Steve replied, "And the really good news is—you never have to buy film for it! That's a saving."

Curt shook his head. "You can see who's the bachelor around here," he said.

"Marry in haste, repent at leisure," said Steve.

"Well, show us what it can do, Steve," said Hanna.

Steve looked around for a suitable subject and then said, "Look, there's Boxie." The cat from the neighboring house often visited for the saucer of milk and tidbits she knew she could wheedle from Hanna. At the moment, Boxie was doing her best to melt into a wooden bench in the morning sunshine. She looked up sleepily as Steve approached her with the camera.

"Got it!" said Steve. He connected the camera to the serial port of his ThinkPad with a thin cable and brought up the camera software. Then he started to download the image. "This will go fast. I've only got one image in the camera," Steve said. A series of small frames was presented on the screen. Only one contained an image, and when Steve double-clicked on it, up came an image of Boxie. Steve zoomed in [see Figure 32].



Figure 32. *Boxie the Cat*
The real picture is in full color!

“Oh, that’s lovely!” said Hanna. “I’ve often wanted a picture of Boxie, but I’ve never gotten around to bringing in my camera. Could you print that, Steve?”

“Sure thing,” said Steve. He selected the print menu, chose the color printer, and clicked on the OK button. A short while later the printer oozed out a picture. They picked it up and inspected it.

“This is pretty good, Steve,” said Hanna.

“Yes, and you can do a lot with the image before you print or store it,” Steve responded. “You can rotate it and crop it. You can edit the color tones too, very simply. This picture has come out a bit blue. It would be easy to warm it up by emphasizing the red tones. Anyhow, I’m taking this camera out to Classy Cars to show them how they could capture multimedia images. I’m sure they’ll be excited. Which brings me back to the question of how we’re going to build the code.”

“Maybe Curt could look at that while you’re busy,” said Hanna. “If you swap ThinkPads with him, he can get on with it while you’re out visiting Classy Cars’ branches.”

“Oh—aren’t you still busy with Trusty Trucks, Curt?” asked Steve.

“Not unless something breaks,” answered Curt. “If you’ll let me have your multimedia ThinkPad, I’ll give it a whirl. You keep on telling us how easy it is to handle multimedia in REXX. And the BLOB support in DB2 Version 2 should make it easy to store and fetch multimedia data.”

“OK,” Steve agreed somewhat reluctantly. “I’ll have to transfer the files I need this week onto your ThinkPad. I’ll upload them onto the server.” Steve sat down again, powered on, and plugged his PC into the LAN. Curt, likewise, started uploading files from his PC onto the server.

“Wait a minute,” said Steve. “I’ve built up a whole set of folders with special icons for the Classy Cars project and I don’t want to have to rebuild them all on your PC, Curt.”

“No problem, Steve,” responded Curt. “Just drag them and drop them on the server directory icon. So long as you keep the <Ctrl> key pressed, OS/2 will copy the whole structure for you. Then you can drag the icon off the Server and drop it onto my PC’s desktop.”

“Of course!” said Steve. “Great system, OS/2.”

They swapped ThinkPads and downloaded their files from the server. Steve powered off Curt’s PC, put it in his bag, and left to accompany the Classy Cars IT Manager on a series of visits to their bigger remote branches.

Curt opened the folders that Steve had defined for the Classy Cars project and tried running the multimedia demo. He was able to display images and play audio and video clips. He opened the settings notebooks of the various icons to find out what REXX commands they used, and opened these commands in the editor. “This does look quite straightforward,” he said. “Now I’ll need to dig into the DB2/2 manuals and find out how BLOBs work.”

“Let me know when you find out,” said Hanna, looking up from her work. “I’d like to understand more about BLOBs too.”

Using DB2 BLOBs from Object REXX

Curt spent the next several hours reading the DB2/2 manuals and building small pieces of code to try out the BLOB features. By midafternoon he was ready to share with Hanna something of what he had learned.

“The DB2 developers have done a great job with BLOBs,” said Curt. “I’ve managed to get some things working without any trouble at all.”

Hanna closed her ThinkPad’s lid and came over to see what Curt had built.

“For starters,” said Curt, “they have defined three different types of BLOBs. Well, LOBs, actually—large objects, they call them. Binary LOBs are just one of these three types. They have also defined Character LOBs [CLOBs] and Double-Byte Character LOBs [DBCLOBs] too.”

“If we’re storing images and audio and video clips, we’ll need just plain BLOBs, right?” asked Hanna.

“Yes,” said Curt. “We can define multiple LOBs in a single row, and each LOB can be up to 2 GB in size.”

“That’s huge!” gasped Hanna. “How do you go about loading LOB data into a DB2 column?”

“Well, you can assemble the LOB data into a host variable and then put that into a DB2 column with a normal SQL insert or update statement,” Curt answered. “Or if the LOB data is in a disk file, you can simply give DB2 a host variable that contains the name of the file that contains the LOB data on disk. That way the application program doesn’t need to read the entire LOB into storage. DB2 copies the LOB data straight from its source disk file into a DB2 column.”

“That’s a nice option,” said Hanna.

“Yes,” agreed Curt. “I’ve defined a simple DB2 table and written a load program in Object REXX that loads a BLOB into it. I just pass DB2 the name of the file that contains the BLOB. Look, here it is.”

Curt showed Hanna the code [see Figure 33]. “See,” he explained, “I need to tell DB2 that my host variable is a locator and contains a file name. I declare it with the *language type blob file* options. If I were coding this in C, COBOL, or Fortran, I would have to build a structure containing information about the file—its name, length, and whether I wanted to read it or write it. The REXX interface is much simpler. My locator host variable is actually the name of a REXX stem variable. I store the name of the file in a compound variable, using the stem with the *name* tail, and store the file read/write options using the *file_options* tail. This is the code I wrote to declare the file locator and the update statement that transfers the media file into the DB2 *picture* column,” said Curt.

```

1 call sqlexec 'declare :media language type blob file'
2 media.name = 'd:\cardeal\media\boxie.bmp'
3 media.file_options = 'read'
4 stmt = 'update myTable set myBLOB = cast(:media as blob(4M))'
5 call sqlexec 'prepare s1 from :stmt'
6 call sqlexec 'execute s1'
7 call sqlexec 'clear sql variable declarations'
```

Figure 33. Using REXX to Update a DB2 BLOB

“Look,” said Curt, “to update the DB2 BLOB:

- 1 “I declare a file locator variable called *media*.”
- 2 “I put the name of a bitmap image file into the *media.name* variable, and
- 3 “I put the file options into *media.file_options*.”

“OK so far,” responded Hanna. Curt continued, “Then,

- 4 “I build an SQL statement to update *myBLOB*—the column that contains the BLOB—and next
- 5 “I prepare it,
- 6 “I execute it, and
- 7 “I clear the SQL variable declarations.”

“Hold on,” said Hanna. “What’s the *cast(:media as blob(4M))* in line 4 for?”

“*Cast* is new in Version 2,” replied Curt. “I used it here to tell DB2 that my BLOB file locator host variable *media* will be used to store a BLOB no bigger than 4 MB.”

“And the *clear SQL variable declarations* in line 7?” asked Hanna.

“BLOB file locator host variables don’t get released until the process that created them terminates,” said Curt. “Since they’re potentially very big, it’s good practice to release them as soon as possible. That’s what this new *clear* command does.”

“Well that took some explaining, but you managed to get the job done with very little code,” said Hanna. “Is it just as easy to get the BLOB back out of DB2?”

“It sure is,” replied Curt. “This is the code I developed to do the job” [see Figure 34].

```

1 call sqlexec 'declare :media language type blob file'
2 media.name = 'media.bmp'
3 media.file_options = 'overwrite'
4 stmt = 'select myLOB from myTable'
5 call sqlexec 'prepare s2 from :stmt'
6 call sqlexec 'declare c2 cursor for s2'
7 call sqlexec 'open c2'
8 call sqlexec 'fetch c2 into :media :mediaInd'
9 call sqlexec 'close c2'
10 call sqlexec 'clear sql variable declarations'
```

Figure 34. *Using REXX to Fetch a DB2 BLOB*

“First,

- 1 “I declare the *media* file locator variable. Then
- 2 “I put the name of a bitmap image file into the *media.name* variable, and next
- 3 “I put my *overwrite* file option into *media.file_options*.”

“This looks familiar,” responded Hanna.

“Yes, I was able to copy some of this code from the update program,” Curt agreed.

4 “This is the SQL select statement. Very simple! I have only one BLOB loaded, so I don’t even need to specify which row. Of course, the code I develop for Classy Cars would specify a vehicle in this select statement. Next,

5 “I prepare the SQL select statement,

6 “I declare a cursor on it,

7 “I open the cursor and

8 “I fetch the BLOB into the file.

9 “I close the cursor and

10 “I clear the SQL variable declarations.”

“And then?” asked Hanna.

“And then it’s ready to display,” answered Curt. “Like so.” He opened the OS/2 *Drives* folder, found the image file that DB2 had just created, and copied it into the OS2\Bitmap directory. Then he dragged a new folder from *Templates*, opened its settings, and changed its background to the bitmap that he had just copied into the OS2\Bitmap directory. A picture of Boxie appeared in the folder.

“That’s great, Curt!” said Hanna. “And it didn’t take you long at all. What more do you have to build?”

“I need to change the Classy Cars database definitions,” said Curt. “Apart from adding a new column to the vehicle table, I have to define a separate DB2 table space to store the multimedia data. Then, when I define the new vehicle table, I’ll specify that DB2 should perform no logging on the column that holds the multimedia data. Otherwise, it would waste log space [see Figure 35].”

```
CREATE REGULAR TABLESPACE VEHICLESPEACE          -- table space for non multimedia
MANAGED BY DATABASE                                -- columns of the vehicle table
USING ( FILE 'vehiclea' 300);                      -- 300 blocks of 4K

CREATE LONG TABLESPACE VEHICLESLOB                -- table space for long (BLOB)
MANAGED BY DATABASE                                -- columns of the vehicle table
USING ( FILE 'vehicleb' 2000);                     -- 2000 blocks of 4K = 8 MB

CREATE TABLE CARDEAL.VEHICLE                      -- Vehicle table
(SERIALNUM          INTEGER    NOT NULL,
CUSTNUM             SMALLINT   NOT NULL,
MAKE                CHAR(12)   NOT NULL,
MODEL               CHAR(10)   NOT NULL,
YEAR                SMALLINT   NOT NULL,
PICTURES            BLOB(4M)   NOT LOGGED ) -- BLOB column, up to 4 MB
IN VEHICLESPEACE      -- assignment for normal columns
LONG IN VEHICLESLOB;  -- assignment for long (BLOB) column
```

Figure 35. DB2 Definition for the Vehicle Table with Multimedia

“Then, of course, I have to generalize this code to support multiple media files per vehicle, and also different media types. Currently, I’m handling only images, but audio and video will be almost identical. I guess I should have something working by the end of the week.”

“Wonderful!” said Hanna. “Steve will be back in the office on Monday and then we can all look at it together.”

Multiple Multimedia Files in BLOBs

“If you’re planning to store images and audio clips jumbled together in one BLOB, how on earth will you ever unscramble them?” asked Steve. It was Monday, Steve’s first morning back at the office in a week.

“No problem!” said Curt with a smile. “DB2 provides facilities to break BLOBs into pieces and handle one piece at a time. The really neat thing about this is, DB2 doesn’t even have to read the whole BLOB into its own buffers to give you access to the part you need. That’s very important. The images and audio clips may range from 100 to 500 KB each, but if we include video clips ...”

“Are video clips for real, Steve?” interrupted Hanna.

“Sure!” replied Steve. “We can do only limited video on this ThinkPad, but the latest home PCs from IBM include a chip that can display full-screen, full-motion video. And, of course, the IBM PowerPC is so powerful that it can deliver full video without needing any hardware assist. It does the whole job in software.”

“Wow!” said Hanna.

“It’s true,” agreed Curt, “and I’ve heard that the Intel Pentium Pro processor has the same kind of capability.”

“Yes,” said Steve, “and it would be silly for us to ignore that kind of capability, since it’s almost here now. And our car dealer application is going to be around for a *long time*, isn’t it, team?”

“Right!” chorused Hanna and Curt.

“So it won’t hurt to make sure that our multimedia design can handle video when our customers ask for it,” continued Curt. “OS/2’s multimedia capabilities make it very easy to handle video. It uses the same commands as audio. And it sure makes for a powerful demo when you’re trying to close a sale!”

“That sounds great, Curt,” said Steve, “but you haven’t answered my original question. How are you going to separate out all this multimedia data if you jumble it together in one big BLOB? Wouldn’t it be better to store each piece of multimedia data as a separate BLOB? You could add three new columns to the vehicle table—one for the image, one for the audio, and one for the video.”

“DB2 would allow us to do that,” replied Curt, “although some other database managers wouldn’t. But Classy Cars wants to be able to store several pictures of some of their cars—front view, side view, and so forth.”

“Oh, yes,” said Steve. “Well, it’s obviously a repeating group, so why don’t you normalize the data? Create a new DB2 table called *VehicleMedia*, key it on the vehicle’s serial number and a multimedia sequence number, give it a BLOB column, and put the multimedia descriptive data in there. That way each vehicle could have as many or as few associated multimedia files as you want.”

Curt looked thoughtful. “That approach could also work, Steve,” he said, “but if you’d just listen for a moment, I’ll tell you how I’m handling it.”

“OK,” said Steve, resolving to be patient.

“I’ve written code to put all the multimedia data for a given car into one BLOB. I also embed control information in the BLOB. I use the first 3 bytes of each BLOB to store a counter that tells me how many multimedia files it contains. It’s in character format, so that allows me up to 999 multimedia files per vehicle.”

“OK so far,” said Hanna, “tell us more.”

“Following the 3-byte counter,” Curt continued, “I’ve got a 30-byte string of control information for each multimedia file in the BLOB. It contains a 20-byte title for the multimedia data, and its length. I strip these out using the DB2 substring function and pass them to the GUI code, which inserts the titles into a list box. This shows the user which multimedia files are available for playing. I use the relative position of the title within the list and the sizes of the multimedia files that come before it to calculate its position within the BLOB. Then I use the DB2 substring function to pull just the bytes we need out of the BLOB. So when the user clicks on a particular title and asks to play it, it’s very efficient.”

“Does DB2 read the multimedia data into one of your program variables?” asked Hanna.

“It could,” answered Curt, “but I’m exploiting DB2’s ability to transfer the data directly from the BLOB to a file on disk. My program never even sees the data. Once DB2 has copied it to disk, I issue a multimedia *play* command, and the rest happens automatically.”

“How do you load multiple multimedia files into one DB2 BLOB column, Curt?” asked Steve. “Do you read each file into a separate REXX variable, concatenate them together in storage, and then insert that data into the DB2 column?”

“I thought of doing it that way,” answered Curt, “but then I found an easier way. This is how I build the SQL that I need.” Curt showed them a piece of code [Figure 36].

```

1  hostvar = ':ctlinfo'
2  ctlinfo = right(numpic,3):'.'
3  stmt = 'update cardeal.vehicle set pictures = cast(? as blob(1K))'
4  do i=1 to numpic
5      piclength = stream(picfile.i,'c','query size')
6      ctlinfo = ctlinfo'left(picfile.i,20)','right(piclength,8)';
7      call sqlexec 'declare :vpic'i 'language type blob file'
8      hostvar = hostvar', :vpic'i
9      call value 'vpic'i'.name', picfile.i
10     call value 'vpic'i'.file_options', 'READ'
11     stmt = stmt '|| cast(? as blob(4M))'
12 end
13 ctlinfo = "BIN"ctlinfo"@"
14 stmt = stmt 'where serialnum =' oldserial
15 call sqlexec 'prepare s1 from :stmt'
16 call sqlexec 'execute s1 using' hostvars

```

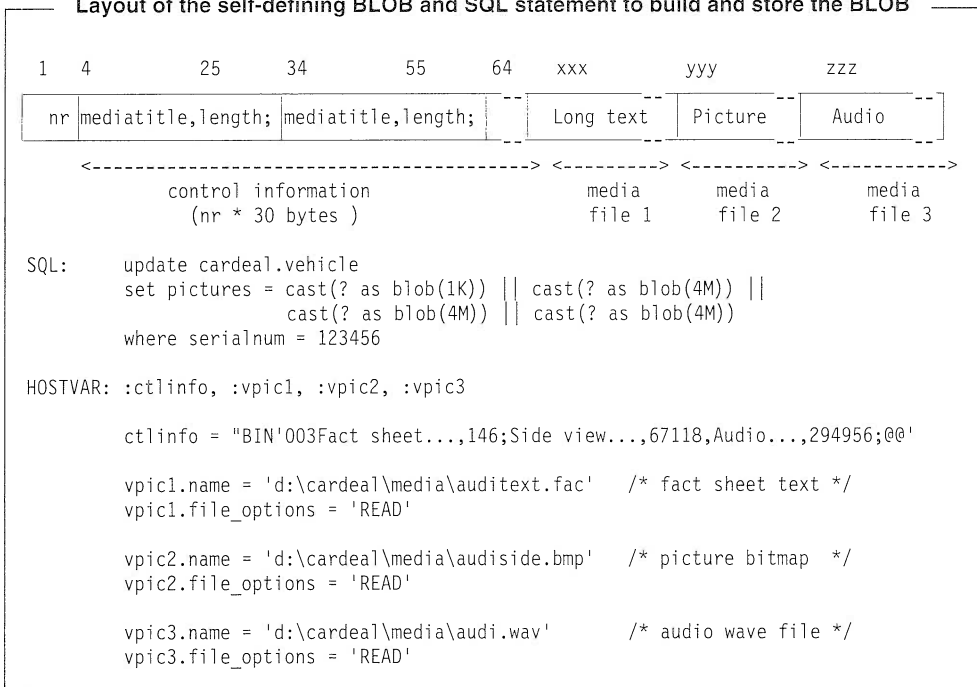
Figure 36. *Using Object REXX to Build and Store a DB2 BLOB*

Curt explained his logic. “To begin,

- 1 “I start the list of host variables with the control information variable.
- 2 “I initialize this variable with the number of media files.
- 3 “I code the beginning of the SQL update statement.
- 4 “I loop as many times as there are media files to be inserted into DB2. Each time,
- 5 “I get the length of the media file, and
- 6 “I concatenate the title and length to the control information.
- 7 “I declare a new DB2 locator file host variable and
- 8 “I concatenate its name to the list of host variables.
- 9 “I set the locator variable’s file name and
- 10 “I also set its file options.
- 11 “I concatenate another place-marker to the SQL update statement.
- 12 “Once out of the loop,
- 13 “I mark the control information host variable as binary, and
- 14 “I complete the SQL update statement.
- 15 “I prepare it, and
- 16 “I execute it, using the list of host variables that I built up.

“DB2 concatenates all the multimedia files together to form a single BLOB field, and I never even touch them in my Object REXX code. Pretty neat, hey?”

Layout of the self-defining BLOB and SQL statement to build and store the BLOB



“Wow! I never realized how powerful DB2’s BLOB handling capabilities are,” said Hanna. “And you’re making full use of them, Curt.”

Curt smiled. “Thanks, Hanna. Want to hear it play?”

“Yes please,” said Hanna.

Curt started up the application using the Dr. Dialog interface that Steve had developed. “I’ve added the logic for the button called **Media**, under the vehicles list box,” he said. “You select a vehicle, then click on the **Media** button. The application opens a new window, which shows a list of the multimedia files available for the currently selected vehicle, if any.” Curt opened the Vehicle Multimedia window as he spoke [see Figure 37 on page 122].

“Then you just click on the multimedia file you want!” Curt did this. A line of text in large bold letters scrolled smoothly from right to left across a yellow area in the Media window. It carried a description of a car.

“And since Object REXX supports concurrent processing,” continued Curt, “I can kick off something else while the text display is still rolling.” He clicked on a multimedia line describing a bit map picture, and a picture of a car appeared in another area of the window. Then Curt selected a sound bite and they heard a recording of his voice describing the car that he had selected.

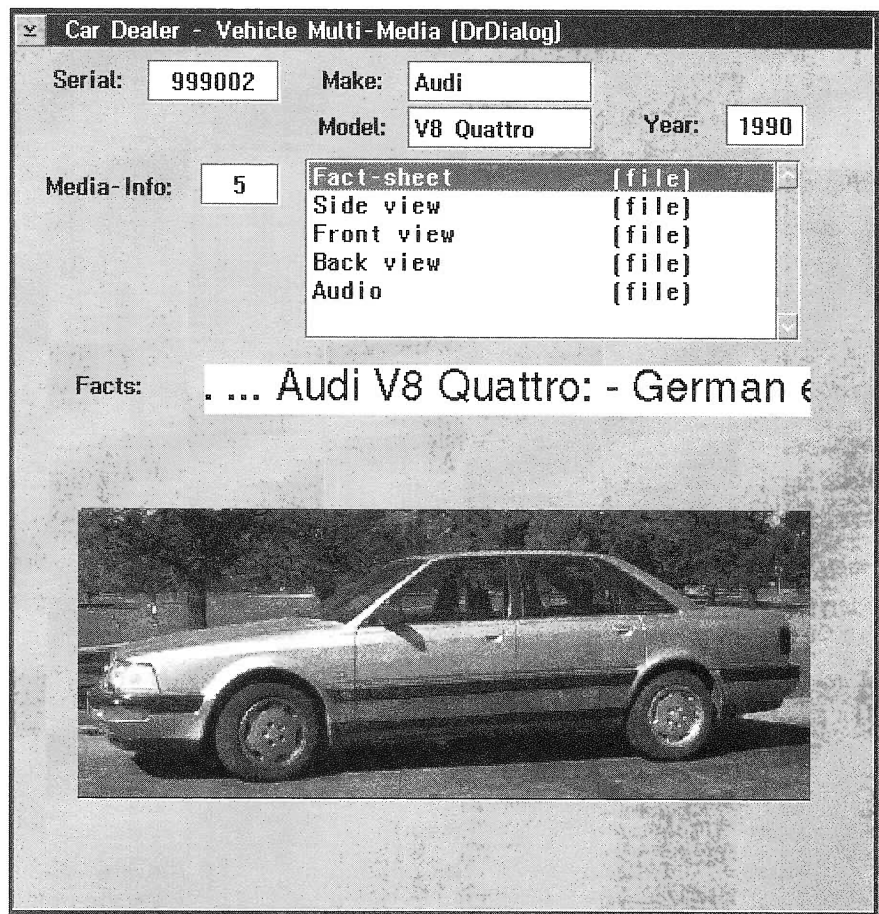


Figure 37. *Vehicle Multimedia Window of Dr. Dialog GUI Application*

“Classy Cars will be really impressed when they see what you’ve developed,” said Hanna. “It will give them a wonderful marketing aid. I know they were thinking of using it for their marketing staff, but it’s so impressive, I think they could also show it directly to prospective car buyers. What do you think, Steve?”

All this time Steve had been watching Curt’s demo silently. “Yes, it is impressive,” he answered Hanna.

“But?” prompted Hanna. She could see that Steve wasn’t completely happy.

“I’m worried about storing multiple multimedia files and the catalog of multimedia information in the BLOB itself,” Steve answered. “We may need to add more control information later on, and this approach is limiting.”

"There's no limit," said Curt. "I can easily increase the size of the control field if we need to."

"Well," responded Steve, "if you had created a new vehicle multimedia table and stored one multimedia file per BLOB, we could add new columns to that table any time we needed to by using the DB2 *ALTER TABLE* command. We wouldn't have to unload the existing data and reformat it, or even change existing applications. They would keep on working, while new applications made use of the new columns."

"What new columns?" asked Curt. "Are you changing the application specs while I'm still writing the code?"

"I was talking with the consultant that Classy Cars has engaged to help them develop an IT architecture," answered Steve. "He isn't happy with our proposal that Classy Cars install a separate database manager in each of their branches."

"Why not?" asked Hanna. "We've costed it out, and it's a good, economic solution. DB2 is inexpensive, and they won't have to get into the complexities and costs that networking their branches into a central database manager would entail. And Classy Cars told us right from the beginning that each branch runs as an autonomous unit. They don't have any need to share data."

"That is what they told us," agreed Steve. "Of course, their head office does want sales and service revenue figures on a weekly basis..."

"We agreed with them that their head office would get that data by dialing into the distributed DB2 databases each week," interrupted Curt.

"Right," agreed Steve. "But the consultant has uncovered something that Classy Cars didn't tell us. They deal with several large companies all over the country. Currently, each branch of Classy Cars deals with the branches of these large companies in isolation. Many of these companies are unhappy with this situation. They want a single, consolidated bill from Classy Cars each month, and they want a single national phone number where they can talk to one person about all their dealings with Classy Cars. The consultant says the best way of achieving this would be to have a single central database manager, with all the branches hooked into it."

"That sounds like a big change, Steve," said Hanna. "I don't know if we can support that approach. What are the implications for our design and the code we have already written?"

"No problem!" interjected Curt. "DB2/2 can support a distributed operation over a wide area SNA or TCP/IP network, and the program code doesn't have to change a bit. All we have to do is install the DB2 OS/2 client package on each remote PC and point them to the central server."

"That's true," agreed Steve. "DB2 gives us a great deal of flexibility in that area. But I'm worried about the implications of shipping multimedia files over the wide area network every time a dealer wants to display a picture or play a sound clip. It would overload their network and kill their response times."

"Well for Pete's sake!" said Curt, his anger boiling up. "It was *your* idea to add multimedia to this application. Classy Cars never even dreamed of doing it till you talked them into it. And now it turns out to be unaffordable, and they'll probably decide to can the whole car dealer application. When will you learn to be sensible and do what the customer asks for, instead of getting so smart you can't deliver what you promise?"

Steve was about to respond angrily, but Hanna stepped between the two of them. "Hold on there, guys," she said. "Let's make sure that we understand how big a problem this is before we start shouting at each other." She held her ground until Curt and Steve backed off.

"It's clearly late in the day for Classy Cars to decide that they want to run on a centralized database," she said. "They've already signed off our design, and that specified one database per branch. If they want to change their minds, we will have to assess the impact of that change and tell them what it will cost, in terms of network bandwidth, extra coding, or whatever. What would the impact be if we ran the multimedia off ASCII files on the users' PCs instead of using DB2 to store them?" she asked Steve.

"That would work fine," he said, "except there wouldn't be any easy way to distribute new multimedia files when new models come out. If we store all the multimedia files in DB2, it's easy to make sure every user has the most recent multimedia information, whether they're local or remote users."

"You can't have it both ways," said Curt. "You can't plan to use DB2 to distribute multimedia files and then complain that it will use too much bandwidth."

"Maybe we can," said Steve. "Supposing we use DB2 to distribute the multimedia data, but keep copies on the users' PCs and reuse them as long as they're still current."

"How would you know if a user's copy is still current?" Curt asked.

"By putting extra columns into the vehicle table," Steve answered. "If we put the multimedia file's time and date stamp in there, our multimedia playing logic could fetch those columns from DB2 and check whether the user has a current copy of the multimedia file on the PC. If not, we ask DB2 to give us a copy, play it to the user, but also keep it for next time. But the way you're handling the multimedia control information, there's no way to do that."

"I could easily extend my control information to handle file date and time stamps," retorted Curt.

“Sure!” said Steve sarcastically. “But what about the next change that we need?”

“Steve, you’re coming up with a whole lot of new requirements and criticizing Curt because his code can’t handle them,” said Hanna. “Now, let’s think this through together. We have already agreed with Classy Cars that we’ll implement our application as a pilot in their San Jose branch next month. That’s going to be six PCs on a LAN with a stand-alone database, right?”

“Yes,” agreed Steve.

“Fine,” said Hanna. “Curt’s multimedia code would work perfectly in that environment, wouldn’t it?”

“Yes,” said Steve again.

“The pilot is due to run for a month,” continued Hanna. “That will give Classy Cars time to think through whether they really need a centralized database and us time to think through the implications of making such a change. Right?”

Steve looked relieved. “You’re right, Hanna,” he said. “I guess I don’t have to start panicking yet. There’s still a fair amount of time before they go live across multiple branches. And if the pilot is successful, their first payment is due. At least we’ll be able to eat while we’re working out what to do next.”

“Now you’re talking, Steve,” said Hanna. “Curt has put together some really smart code, and it seems to work well. We’ve still got time to put it through acceptance testing with the users and implement it as part of the pilot installation. That way, we’ll be delivering it before the date we committed to. I’m sure that Classy Cars will be happy.”

“I sure hope so,” said Curt.

“I’m due back at Classy Cars tomorrow to start working out the implementation plan once the pilot has proved successful,” said Steve. “During the pilot, I’ll talk to them about the approach we want to take for distributed databases.”

“Let us know what happens,” said Hanna.

Implementing the DB2 Multimedia Support

Here are the steps required to implement multimedia in DB2/2 Version 2:

1. Define two DB2 table spaces for the vehicle table, to separate the normal data from the large multimedia data (BLOBs).
2. Define the vehicle table so that the BLOB column is stored in the special table space for such columns (*long in keyword*).

These two steps are shown in Figure 35 on page 117.

3. Write a DB2 program, load-mm.cmd, to update the vehicle table with the multimedia files in the BLOB column. An extract of the program with the SQL statements is shown in Figure 36 on page 120.

- We described all the multimedia files in a specification file named MEDIA.DAT:

```
/* serial, title of file      , filename */
999001, Fact-sheet          , ford.fac
999001, Side view           , fordsid.bmp
999001, Front view          , fordfrt.bmp
999001, Back view           , fordbck.bmp
999001, Angle view          , fordang.bmp
999001, Audio               , ford.wav
999002, Fact-sheet          , audi.fac
999002, Side view           , audisid.bmp
999002, Front view          , audifrt.bmp
999002, Back view           , audibck.bmp
999002, Audio               , audi.wav
...
end
```

- The multimedia update program reads this file and updates the vehicle table with multimedia BLOBs.
4. Write three new methods for the vehicle class to retrieve the multimedia data:

- Retrieve and return the number of media files of a vehicle:

```
::method getmedianumber
  expose medianumber mediacontrol
  if symbol("medianumber") = 'VAR' then return medianumber
  medianumber = 0
  mediacontrol = ''
  stmt = 'select substr(v.pictures,1,3)' ,
        ' from cardeal.vehicle v where v.serialnum = ' self~serial
  call sqlxec 'PREPARE s2 FROM :stmt'
  if sqlca.sqlcode \= 0 then return 0
  vpicind = -1
  call sqlxec 'DECLARE c2 CURSOR FOR s2'
  call sqlxec 'OPEN c2'
  call sqlxec 'FETCH c2 INTO :vpic :vpicind'
  call sqlxec 'CLOSE c2'
  if vpicind >= 0 then medianumber = vpic
  return medianumber
```

- Retrieve the control information of multimedia files for a vehicle:

```
::method getmediacontrol
  expose medianumber mediacontrol
  if symbol("medianumber") = 'LIT' then return ''
  if medianumber <= 0 then return ''
  stmt = 'select substr(v.pictures,5,30*' medianumber ')' ,
        ' from cardeal.vehicle v where v.serialnum = ' self~serial
  call sqlxec 'PREPARE s2 FROM :stmt'
  call sqlxec 'DECLARE c2 CURSOR FOR s2'
  call sqlxec 'OPEN c2'
  call sqlxec 'FETCH c2 INTO :vpic :vpicind'
  rcv = sqlca.sqlcode
```

```

call sqlxec 'CLOSE c2'
if rcv = 0 & vpicind >= 0 then mediacontrol = vpic
return mediacontrol

```

- Retrieve one multimedia file from the BLOB of a vehicle:

```

::method getmediainfo
  expose medianumber mediacontrol
  if symbol("medianumber") = 'LIT' then return ''
  if mediacontrol = '' then self~getmediacontrol
  arg medianum
  if medianumber = 0 | medianum > medianumber | medianum <= 0 | ,
    mediacontrol = '' then return ''
  mediatitle = substr(mediacontrol,medianum*30-29,20)
  medialength = substr(mediacontrol,medianum*30- 8, 8)
  mediastart = 7 + 30 * medianumber
  do i=1 to medianum -1
    blg = substr(mediacontrol,i*30-8,8)
    mediastart = mediastart + blg
  end
  call sqlxec 'CLEAR SQL VARIABLE DECLARATIONS'
  call sqlxec 'DECLARE :vpic3 LANGUAGE TYPE BLOB FILE'
  vpic3.file_options = 'OVERWRITE'
  temp = value('TMP',, 'OS2ENVIRONMENT')
  if temp = '' then temp = directory()
  tnam = 't' self~serial 'medianum
  select
    when mediatitle = 'Fact-sheet' then vpic3.name = ''
    when mediatitle = 'Audio'      then vpic3.name = temp\\'tnam'.WAV'
    when mediatitle = 'Video'     then vpic3.name = temp\\'tnam'.AVI'
    otherwise                     vpic3.name = temp\\'tnam'.BMP'
  end
  vfacts = vpic3.name
  stmt = 'select substr(v.pictures,'mediastart','medialength')' ,
        ' from cardeal.vehicle v where v.serialnum = ' self~serial
  call sqlxec 'PREPARE s2 FROM :stmt'
  call sqlxec 'DECLARE c2 CURSOR FOR s2'
  call sqlxec 'OPEN c2'
  if vfacts = '' then call sqlxec 'FETCH c2 INTO :vfacts'
  else call sqlxec 'FETCH c2 INTO :vpic3 :vpicind3'
  if sqlca.sqlcode \= 0 then vfacts = ''
  call sqlxec 'CLOSE c2'
  call sqlxec 'CLEAR SQL VARIABLE DECLARATIONS'
  return mediatitle::'vfacts

```

The fact sheet is retrieved directly into a variable, whereas pictures (.bmp), audio (.wav), and video (.avi) are retrieved into temporary files (in the OS/2 *SET TMP* directory), and the file name is returned to the caller.

5. Write the code to play audio and video multimedia files. Sample code to play an audio file:

```

::method playaudio class
  arg filename
  call mciRxSendString 'open waveaudio alias audio shareable wait', 'RetSt', '0', '0'
  call mciRxSendString 'load audio' filename 'wait', 'RetSt', '0', '0'
  call mciRxSendString 'set audio time format ms', 'RetSt', '0', '0'
  call mciRxSendString 'play audio wait', 'RetSt', '0', '0'
  call mciRxSendString 'close audio wait', 'RetSt', '0', '0'
  call mciRExit

```

The code for video is similar, with the *waveaudio* keyword replaced by *digitalvideo* and the *audio* keyword replaced by *video*.

6. Pictures are displayed using PM controls provided by the visual builders.

Implementation Notes

1. The DB2 table spaces and the vehicle table are set up by the installation program. All tables are loaded as well, including the multimedia data. We have provided the programs `runsql.cmd`, `load-db2.cmd`, and `load-mm.cmd`; similar code is embedded in the installation program.
2. Multimedia data cannot be loaded for DB2 Version 1. The rest of the application is fully functional, however. It is still possible to see multimedia in action, because we added this support to the FAT implementation, as well (see below).
3. We retrofitted multimedia support into the FAT implementation. The FAT vehicle class was enhanced by the same three methods (*getmedianumber*, *getmediacontrol*, *getmediainfo*), and the respective multimedia files are passed to the code directly from the distributed data files.

This allows the ASCII and GUI applications to be run with FAT persistence support and multimedia data.

4. Multimedia data is available for customers *New and used cars*, *Furukawa*, *Griborn*, *Turton*, and *Wahli*.

Source Code for DB2 Multimedia Implementation

The source code for the DB2 Vehicle class is listed in *DB2 Vehicle Class* on page 319.

The multimedia descriptive file is listed in *Multimedia Data Definition File* on page 292.

Audio and video play methods are part of the car dealer class, which is introduced in *Overall Car Dealer File Structure* on page 150, and the source code is listed in *Cardeal Class* on page 308.

The source code for the multimedia load program is not listed in the appendix; it is available in the car dealer directory on the CD or after the sample applications have been installed on a hard drive (see Table 35 on page 259).

9

Data Security with Object REXX and DB2

In this chapter we exploit DB2 stored procedures to solve a common security problem when using dynamic SQL.

The Security Problem

The next day, Hanna and Curt were working quietly in the office when the phone rang. Hanna answered it.

“Hello, Hacurs Software Systems,” she said. “Oh hello, Steve. We were wondering how your morning went with Classy Cars. What? What’s the problem? They don’t like our data security? No ...wait ...hold on, Steve! I think that you’re overreacting. Come into the office straight away and tell us what happened. I’m sure we’ll be able to sort something out.”

“That sounded like trouble,” observed Curt.

“Steve’s very upset,” said Hanna. “He said that the IT consultant that Classy Cars has engaged has persuaded them that there’s a major security exposure in our system. Steve said he couldn’t talk them out of it; they’re saying they can’t implement the system as it is.”

“We’ve invested a lot of effort in Classy Cars, and so far we’ve had nothing but problems with them,” said Curt. “I think we should cut our losses. There are lots of other car dealers around. We’re sure to sell our system with less time and trouble than we’re having.”

“We’ve invested too much time and effort in Classy Cars to just walk away,” Hanna responded. “Let’s wait and see what the problem is, Curt. I’ve got a lot of faith in you and Steve. I’m sure you’ll be able to handle it.”

Curt didn’t look convinced. They both tried to keep working at their tasks while waiting for Steve. At last they heard the crunch of his car tires outside the office. He came in moments later, slamming the door.

“That confounded consultant is really making things hard for us,” Steve said.

“What is it now, Steve?” asked Hanna.

“Well, he’s still going on about the need for a centralized database manager,” said Steve. “And he seems to have convinced Classy Cars that that’s the way they must go. I explained to them that they have already signed off our design based on a separate database per branch, and that’s the way we’ll have to install the pilot. It’s too late to change without impacting the schedule. They agreed to that. We’ll investigate the impact of changing to a distributed system later on.”

“What else, Steve?” asked Hanna.

“The consultant has been reviewing our design in detail,” Steve responded. “He’s come to the conclusion that its security is weak, and that this will be a problem, particularly for their customer data.”

“Why does he say that?” asked Hanna.

“Mainly because we’re using dynamic SQL in all our applications,” answered Steve. “The implication is that we will have to authorize the end users to access the DB2 tables directly. So long as they use our programs, we can control what data they can see. But they could equally well use a package like Lotus 1-2-3 or Excel to access the tables, and then we can’t control what they do.”

“But isn’t access to DB2 password-protected, Steve?” asked Hanna.

“Yes,” replied Steve, “but the user is asked to perform the logon when our code first tries to access DB2. So users have to know their own logon ID and password. And once they’ve logged on to allow our application to run, they can start up other applications and access the database with them.”

“Hold on, Steve,” said Curt. “What you say is true for a user working on the database server itself. But someone using a client PC to access a DB2 server must issue a *connect* command to DB2, quoting a user ID and password. That *connect* command must be embedded in the application program. And we’re not planning to allow users to run applications directly on the DB2 server machine.”

“I wish that I’d had you with me this morning, Curt,” said Steve. “But if the DB2 connect statement is in a REXX program, anyone can look at the source, and then they’ll see the user ID and password, won’t they?”

“That used to be true, but not any more,” said Hanna. “Object REXX includes a new utility called *REXXC*. You can use this to read REXX source code and produce a new program that does what the original program does, but is unreadable, similar to compiling C or COBOL source, which produces unreadable object text.” [See *The REXXC Utility* on page 267 for more information.]

“That’s great,” said Steve, “but if a REXX program contains a user ID and password as plain text literals and it gets processed by the *REXXC* command, won’t the literals still be there in the file written by *REXXC*?”

“Let’s find out,” said Curt. He typed in a small REXX command file, processed it through *REXXC*, and looked at the output. “Hmm—yes, the literals are stored as plain text in the output file.”

“Hey!” said Steve. “Maybe we can write the logon data in an encrypted form in our programs, then include code to decrypt it before we pass it to DB2. The Object REXX *translate* or *bitxor* methods could do the trick. That way, no one could see the logon information in our programs, because it wouldn’t be there—in readable form.”

“I’m impressed!” said Hanna. “You guys have thought of lots of ways to tackle this security issue, and it didn’t take you long at all.”

“I’m going straight back to Classy Cars,” said Steve. “I think we can overcome their concerns about security.”

“Let us know how it goes,” called out Hanna as Steve strode for the door.

Hanna and Curt settled down to their work again. It was mid-afternoon before they heard from Steve. He strode into the office with a troubled look on his face.

“Hi, Steve,” said Curt. “How did Classy Cars like our approach to ensuring the security of their data?”

“They were impressed,” Steve answered. “So was their IT consultant, except on one issue. They definitely want to move toward a single, centralized database. They accept that this wasn’t in the original spec that they signed off, so they’re prepared to implement using separate databases in each branch and then to centralize over time. I’m sure we can work out an approach that will satisfy them.”

“What’s the IT consultant’s concern?” asked Curt.

“He’s been helping Classy Cars plan how they will run in the future,” Steve answered. “They want central control over their customer accounts. It turns out that lots of their customers have run up substantial debts, and some simply switched their business to another branch when the first branch refused to extend them any more credit. Their branches don’t exchange customer information. They really want to fix that. But they’re pretty worried, because the accounting data they want to store will be very sensitive. They want a guarantee that unauthorized personnel can’t read it—and, even more important, alter it.”

“I don’t understand the problem, Steve,” said Hanna. “DB2 has very good built-in facilities to restrict the people who can access a specific table.”

“Yes,” agreed Steve, “but every branch must have users authorized to capture and view accounting data for their own branch only, but not to update or delete it. I was trying to work out a way we could do this with the *check* option of the DB2 view facility, but their requirements as to who can see what and who can update what may be too complicated to handle this way. The consultant says that the best way to implement really tight security is to have the application code running on the server in a locked room, and not on a hundred-plus PCs all around the country. He says if you can’t even keep games and viruses off users’ PCs, how can you hope to keep fraudulent code off?”

“So what’s wrong with the security schemes we came up with this morning?” asked Curt.

“Well, our schemes all depend upon the remote user having an access user ID and password,” said Steve. “We can do a lot to keep those hidden, but if there’s collusion and somehow an ID and password pass into the wrong hands, the security of their accounts data will be lost, and they won’t even know about it.”

“Is this for real?” asked Curt. “Aren’t they being a bit paranoid?”

“I’m afraid not,” answered Steve. “Last year they had to write off more than a quarter of a million dollars in bad debt. They suspect that there may have been some collusion between some customers and some of their staff. But there’s no way they can prove anything. They just don’t have the right controls in place. Our computer system could help them, but its security will have to be watertight.”

“Ouch!” said Curt. “It’s a tough world out there.”

“I’ve got an idea that some of the new DB2/2 features may provide a solution in this area,” said Hanna, “but I’m not sure if we can use them from Object REXX. I’ll take the manuals home tonight and check it out.”

“Good hunting, Hanna,” said Steve. “If you can come up with a really watertight solution to their problems, it would be worth a lot to them—and to us!”

Coding Stored Procedures with Object REXX

The next morning, Hanna came in clutching the DB2 manuals and smiling. “I think I’ve got the answer, guys!” she said.

“Cut the suspense, Hanna!” said Steve. “Tell us what your approach is, please.”

“The consultant suggested that the code dealing with the most sensitive data should run on the central server only,” said Hanna. “Historically, that’s the way transaction programs have been handled on mainframes. Normally we build our REXX programs to run directly on the client PCs. Our challenge is to find a way to get some of the REXX code to run on the secure server and still be able to access it from the client PCs. The answer I thought of is to use DB2’s stored procedures facility. It’s often used to reduce network traffic and to improve server performance by moving code that accesses the database heavily onto the database machine. But it can also be used to improve security. It would allow us to move key code off the client’s PCs and to access secure DB2 tables by using a special logon ID and password in code that runs on the server only.”

“Hanna,” said Curt, “I hate to puncture a great idea, but a client program must be connected to a DB2 database before it can invoke a DB2 stored procedure. Which means that it has already supplied a logon ID and password. The DB2 stored procedure isn’t allowed to issue another *connect* command, so it has to operate under the client’s ID and password. Anything that the stored procedure can do in DB2, the client can do, too. And since the client’s ID and password are embedded in code on the client PC, your proposal doesn’t sound any more secure than do the approaches Classy Cars has already turned down.”

Hanna just smiled. Curt and Steve were really intrigued. What did she have up her sleeve? She moved to the whiteboard and picked up the pen.

“What you say is true, Curt,” she said. “I thought of that, too. But I also thought, Is there anything new in Object REXX that could help us in this situation? I did some reading about DB2 and Object REXX last night, and I came up with an idea. I wrote some code to check it out, and it looks like it will work. But I had only one ThinkPad to test it on. We’ll have to try it out on a couple of PCs connected over our LAN.”

“Come on, Hanna, you’re driving us crazy!” exclaimed Steve. “What’s your idea?”

“It’s quite simple, really,” said Hanna. And for once, she actually drew something on the whiteboard. “This is the way that DB2 stored procedures are normally put together,” she said [see Figure 38].

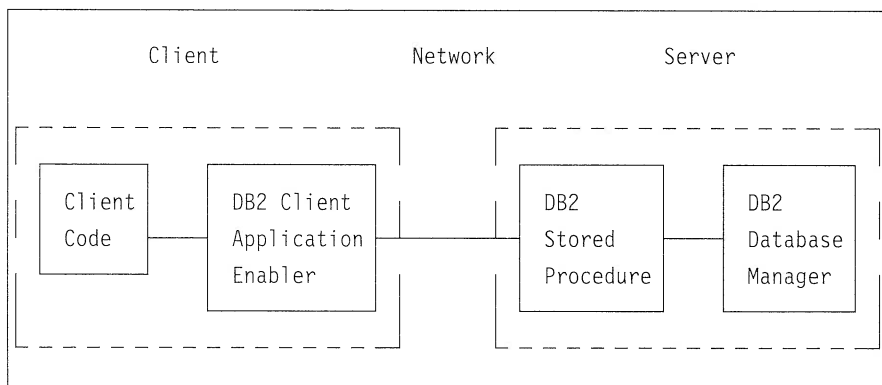


Figure 38. *DB2 Stored Procedure*

“The client code tells DB2 to call a stored procedure,” explained Hanna. “The Client Application Enabler package [CAE/OS2] on the client PC relays the request to the DB2 database manager on its server. DB2 schedules the stored procedure code, passing it the arguments on the original call command. The stored procedure runs, accessing DB2, and passes results back to DB2, which relays them back to the client code.”

Hanna continued. “We can write both the client code and the stored procedure in Object REXX. But as you pointed out, Curt, the stored procedure has to use the DB2 connect that the client code has already issued, and, therefore, has no more authority than does the client. Now for the magic!” Hanna changed the figure on the whiteboard [Figure 39].

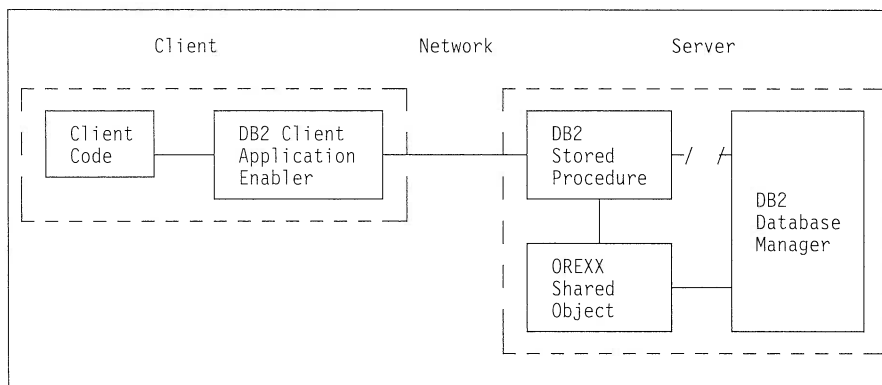


Figure 39. *DB2 Stored Procedure with Object REXX Shared Objects*

“The key thing here is that although the DB2 stored procedure has access to DB2 by virtue of the connect that the client PC issued, *it doesn’t use it*,” Hanna explained. “At least, not for accessing the really sensitive data, because the client doesn’t have authority to do that. Instead, we automatically start up a disconnected process each time the server boots up, running an Object REXX server command. This command issues its own connect to DB2 using a secure ID and password, creates a special shared Object REXX object, and stores its name in the *.environment* directory object. Once it’s there, any Object REXX command that runs on the server can find it and use it. In particular, DB2 stored procedures written in Object REXX can find it and use it to relay requests to the Object REXX server command that created the object. The server command waits for requests, handles them, and sends responses back to the requester—all through the shared object and its methods.”

“This is brilliant, Hanna!” said Steve. “Show us your code.”

“This is the Object REXX server code,” said Hanna, bringing up the code on the screen [Figure 40 on page 136].

“Now this isn’t production-strength code. There isn’t any error-checking in it,” explained Hanna. “I just did enough to make sure that it would work the way I thought it should. Let me step you through the main points:

- 1 “This is the file that would run when the server boots up. It contains definitions for the DB2 server class and methods. I create a single object from this class.
- 2 “I store the object’s name in the global *.environment* directory object. [See *Communication Among Classes* on page 152 for more details on the global and local directories.]
- 3 “I issue an OS/2 logon using a special ID and password, then connect to DB2. This gives me the authority to anything the special ID can do.
- 4 “I go into a loop, sending the *Respond* message to the server object. Each time it waits for a client request, then processes it. I make this loop quit if the client sends an exclamation mark [!] character, just to ease debugging.
- 5 “Here’s where I define the DB2 server class and its methods.
- 6 “The *init* class method is invoked when I create a new object. I set the object’s state attribute to *Free*. The other values it can have are *Request* and *Response*. It controls access to the Object REXX DB2 server object.
- 7 “The Object REXX DB2 server process invokes the *Respond* method to wait for a client request and then process it.

```

/***** server.cmd *****/
server = .DB2server~new /* make a DB2 server object */ 1
.environment['CARDEAL.DB2SERVER'] = server 2
'logon special /p=secret /L' /* logon as special user */ 3
if RxFuncQuery('SQLEXEC') then
    call RxFuncAdd 'SQLEXEC', 'SQLAR', 'SQLEXEC'
call sqlxec "connect to dealerdb"
say "The OREXX server is active..." /* server iis ready */ 4
do until input = '!'
    input = server~Respond /* process a client request */
end
say "The OREXX server is ending."
.environment['CARDEAL.DB2SERVER'] = .nil /* clean-up */
call sqlxec "connect reset"
exit

::class DB2server /* DB2 SERVER CLASS */ 5
::method init /* invoked by the server */ 6
    expose state
    state = 'Free' /* we're ready for a client */
    return
::method Respond unguarded /* invoked by the server */ 7
    expose state input output
    guard on when state = 'Request' /* wait for a client */ 8
    select
        when input~translate = 'CONNECT' then do /** CONNECT ***/ 9
            stmt = "select user from sysibm.systables",
                    "where name = 'SYSTABLES'" /* return userid */
            call sqlxec "prepare s1 from :stmt"
            call sqlxec "declare c1 cursor for s1"
            call sqlxec "open c1"
            call sqlxec "fetch c1 into :output"
            call sqlxec "close c1"
            end
        when input~translate~word(1) = 'CUST' then do /** CUST xxx */ 10
            custno = input~word(2)
            stmt = 'select * from cardeal.customer where custnum =' custno
            call sqlxec "prepare s2 from :stmt"
            call sqlxec "declare c2 cursor for s2" /* return the */
            call sqlxec "open c2" /* customer info */
            call sqlxec "fetch c2 into :custnx, :custname, :custaddr"
            if sqlca.sqlcode = 0 then
                output = custnx:' strip(custname) 'in' strip(custaddr)
            else output = custno': not found'
            call sqlxec "close c2"
            end
        otherwise output = input~reverse /* just to show we're here */
    end
    state = 'Respond' /* signal client to proceed */ 11
    return input
::method Request /* invoked by the client */ 12
    expose state input output
    use arg input
    state = 'Request' /* signal server to proceed */ 13
    guard on when state = 'Respond' /* wait for server */ 14
    state = 'Free' /* ready for next request */ 15
    return output /* give output to client */ 16

```

Figure 40. DB2 Stored Procedure with Shared Objects: Server

- 8** “The *Respond* method is unguarded so it can run concurrently with client requests, but issues a *guard* command to wait for a client, which will change the state to *Request*. [See Chapter 13, *Object REXX and Concurrency*, on page 217 for details on *guard*.]”
- 9** “There’s only token logic in the *Request* method. If the input transaction is a *CONNECT* request, I tell DB2 to select the special SQL value *User*, which is my DB2 connection ID, and return this to the client. That’s just enough logic to verify that I can access DB2 data and to make sure that I’m using the right DB2 connection.”
- 10** “If the input is a *CUST* request, I retrieve the customer from the database and return its information. Otherwise, I simply reverse the input, just so I can see that something happened.”
- 11** “The *Respond* method sets the object’s state to *Respond*, once it has set up the required response in its output attribute. This signals the waiting *Request* method to finish processing the client’s request.”
- 12** “The *Request* method is invoked by the client and runs under the client’s process. It’s guarded by default, so only one client at a time can run it.”
- 13** “The *Request* method sets the object’s input attribute to the request argument passed by the client, then changes the object’s state to *Request* to signal the server process to handle this new request.”
- 14** “Then the *Request* method waits until the server sets the state to *Respond*, by which time the required response is in the attribute output. [See Chapter 13, *Object REXX and Concurrency*, on page 217 for details on the *guard* instruction.]”
- 15** “I set the object’s state to *Free*. We’re ready for the next client request.”
- 16** “I return the server’s response to the client.”
- “And here’s the DB2/2 stored procedure code,” said Hanna [Figure 41].

```

/***** gateway.cmd *****/
server = .environment['CARDEAL.DB2SERVER']
sqlroda.1.sqldata = 'anything' /* touch argument to make DB2 happy */
sqlroda.2.sqldata = server~request(sqlrida.1.sqldata) /* set reply */
sqlca.sqlcode = 0

```

Figure 41. DB2 Stored Procedure with Shared Objects: Gateway

“You’ll see that there really isn’t much to it. It acts as a gateway between the remote client and the Object REXX server code.”

- 1** “I pick up a pointer to the Object REXX DB2 server object from the *.environment* directory object.”

2 “The input transaction is passed to the stored procedure in the *sqlroda.1.sqldata* REXX variable. I pass this to the *Request* method of the server object and store the result that it gives back to me in *sqlroda.2.sqldata*. DB2 will pass this back to the remote client that called my procedure.

3 “I set the *sqlcode* to zero to indicate that the call worked correctly.”

“The gateway is tiny, Hanna,” said Curt, “there are only four lines of code. Is this what they call *middleware*?”

“I guess so,” Hanna replied.

“Where’s the client code?” asked Curt.

“Here it is,” said Hanna, bringing up the code [Figure 42]. “I wrote this as a stand-alone command that I can invoke from the command line. You’ll see that there really isn’t much to the client code either.”

```

/***** client.cmd *****/
'logon humble /p=user /L'           /* logon as normal user      */ 1
call sqlxec "connect to dealerdb"
reply = " " ~left(60)                /* prime reply so DB2 knows */ 2
proc = "gateway.cmd"                /* gateway code on DB2server*/ 3
say "The DB2 client is active; I'm going to use the DB2 server..."
do until reply = "!"                 /* ask for input             */ 4
  say "Give me an argument (any, connect, cust xxx, ! to end)"
  parse pull argument
  call sqlxec "call :proc (:argument, :reply)" /****** call proc *****/ 5
  if sqlca.sqlcode = 0 then
    say "The reply is "reply""
  else
    say "sqlcode =" sqlca.sqlcode", sqlerrmc =" sqlca.sqlerrmc
end
say "The DB2 client is terminating." /* done                      */
call sqlxec "connect reset"

```

Figure 42. DB2 Stored Procedure with Shared Objects: Client

“Let me step you through this code,” Hanna said.

- 1** “I log on to OS/2 using a low-security ID and password, then connect to DB2 with this ID and its authority. On a separate DB2 client PC, these two statements could be combined into one.
- 2** “I’m going to use the SQL *CALL* command. I have to prepare the field in which the reply will come by assigning a representative value to it.
- 3** “My DB2 stored procedure is called *gateway.cmd*.
- 4** “The client loops, asking the user for input until the user keys in an exclamation mark [!].

- 5 “This is the real meat of the code. The client calls the DB2 stored procedure, passing it the user’s input string in argument and the *reply* field for the reply. I check the SQL return code, and if it’s good I give the reply back to the user. Otherwise, I display the SQL code and error message.”

To show them how it worked, Hanna opened an OS/2 command line window and entered the server command. The server code displayed a message saying it was active and waiting for customers. Then she opened a second OS/2 command line window, and entered the client command. The client code also notified them that it was active, and asked for some input. Hanna typed in Hello! and back came !olleH. She did it again, and the same response came back instantly. Then she typed in connect, and back came the answer SPECIAL.

“That’s the server process’s logon ID, not the client’s,” observed Hanna. Then she typed CUST 106, and the response was the name and address of the customer *Helvetia*. Finally, she typed ! into the client command line. Both the server and the client commands terminated in their respective windows.

“Hanna, this is brilliant,” said Curt, “but the client, gateway, and server code are all running on the same PC. How does this scale up?”

“DB2 stored procedures can be called by remote client PCs,” Hanna answered. “The clients can be running under a variety of operating systems, including OS/2, DOS, Windows, and AIX. DB2 supports connections over both LANs and WANs and can handle SNA, TCP/IP, NetBIOS, and IPX/SPX communications protocols. It’s very flexible.”

“Hanna, you’re a genius!” said Steve. “This approach will allow us to implement really tight control over Classy Cars’ sensitive data. I’m due to visit them later today, and I’d really like to take the code that you’ve developed and to review your ideas and code with them. If they’re happy with it, and I’m sure they will be, we can start implementing their pilot branch while we build secure code to handle their accounts data.”

“I’m glad you like it, Steve,” Hanna replied. “This new version of REXX is so powerful, we’re all going to end up looking like heroes!”

“I love it,” said Steve, “and I’m sure that Classy Cars will, too, once I explain it to them. Let me copy the code you developed, and I’ll go there right away.”

Later that day, Hanna and Curt were in the office, waiting to hear from Steve about how his visit to Classy Cars had gone. They had expected him to phone after his visit, but he hadn’t. There was an edge of nervousness in the office as Hanna and Curt tried to keep busy with things that needed doing. But the Classy Cars deal was so important to the future of their company that they found it hard to concentrate on anything else. Suddenly, they heard Steve’s footsteps outside the

door, and moments later he walked in with his ThinkPad slung over his shoulder and a large brown paper packet in his hands. As he put the bag down, it made a clinking sound.

“How did it go?” asked Hanna. She had a suspicion that the bag contained the answer to her question. Steve didn’t answer immediately. He took a bottle of sparkling wine from the bag and placed it on the desk, followed by three glasses.

“Cut the suspense, Steve,” said Curt. “Tell us what happened.”

“They signed.”

“Congratulations,” said Hanna, giving him a hug. Curt got up and shook his hand vigorously, as if he had just announced his engagement.

“Thanks, Hanna. Thanks, Curt,” said Steve. “This has really been a team effort. We did it together. And Hanna, I want to thank you especially for helping us work as a team whenever Curt and I got bogged down in silly arguments.” Having said this, Steve reached once more into the brown paper bag and, somewhat shamefacedly, pulled out a beautiful, if slightly smashed, bunch of red roses, which he handed to Hanna. This gift was so unexpected that she blushed.

“Thanks, Steve,” said Hanna, “these are really lovely.” She put them into a vase and added water while Curt tried to get some more information out of Steve.

“What did they sign for?” asked Curt. “What do they want us to deliver? And when?”

“They want us to install the application we’ve already built and demonstrated to them,” Steve answered. “They want the GUI front end. After all the trouble they put us to, they’ve decided to go with the Dr. Dialog interface after all! They realize that we really can convert our application to another GUI builder if they ever need us to in the future, and they like Dr. Dialog’s price. Oh, and they want the application to run against multiple distributed copies of DB2, one per branch, for the initial implementation. They want us to quote on extending the application to run off a single centralized copy of DB2 in about three months’ time. And, most importantly, they wrote a check to cover our development effort to date. Here it is!”

Steve pulled out the check and showed it to the others. Hanna took it and said, “Well done, Steve! I’ll deposit that in our bank account later today. Our bank manager will be very happy to see it.”

“All this talking,” said Curt, “and you haven’t even offered us a drink.”

“You’re right,” Steve responded, “what am I thinking of?” He loosened the cork, filled the three glasses, and passed them around. “I propose a toast to Hacurs—may it prosper and grow.”

“To Hacurs,” said Hanna and Curt in agreement.

Demonstration notes

- ❑ The code for the three commands is in the *StorProc* subdirectory of the car dealer application.
- ❑ Before running these commands, define the two user IDs, *SPECIAL* as local administrator with password *SECRET*, and *HUMBLE* as regular user with password *USER*.
- ❑ Make sure DB2 is started.
- ❑ Open two OS/2 windows and make the *StorProc* subdirectory the current directory in both windows.
- ❑ Type *server* in one window, and wait for the *ready* message.
- ❑ Type *client* in the other window, and then enter any text, or the special keywords *CONNECT* or *CUST xxx*.
- ❑ The response of the *SPECIAL* user ID when entering *CONNECT* is proof that the code on the server runs under the user ID of the server.
- ❑ Type *!* to end both client and server.

10

Configuration Management with Object REXX

In this chapter, we discuss ways of managing an Object REXX application that is large and has different versions, all of which must be supported concurrently. We shall see how to use Object REXX classes with inheritance and polymorphism to help us achieve these goals. The grouping of related files into subdirectory structures can also help.

Most REXX programmers know that writing small, one-off applications is fun. It's quick to do, and the users are often grateful to get a fast response to their needs. But, of course, there is always the risk that, after a while, the application might become very popular with many users. Suddenly, the code must run in environments never contemplated and, therefore, turns into a maintenance burden.

This is the kind of problem that a software company like Hacurs loves to have. One-off applications yield revenue once only. The real money-spinners are those applications that can be sold to a number of different customers. Inevitably, the operating environment will be different in each location. There may be different database managers to interface with and possibly different GUI packages, as well.

When classic REXX was first designed, no one could have guessed how widely used it would become or how big some REXX applications would grow. The base language has some facilities for managing large applications by separating called subroutines into separate files, but Object REXX brings a lot more to the table.

We've spoken about the benefits of classes, polymorphism, and inheritance. We've claimed that these make it easier to substitute parts without impact to the rest of the system. Now it's time to deliver. We must show practical ways to make these promises come true. In this chapter, we talk about the problems that confront "successful" applications and show how the features of Object REXX can be used to ease the creation, distribution, implementation, and maintenance of such applications.

Breaking an Application into Multiple Files

Hanna was working alone in the Hacurs offices. Curt and Steve were out at Classy Cars, training more staff to use the pilot car dealer application that was now installed and running. Hanna was working through the various pieces of the car dealer application on the server. Although it had been Hacurs' goal to work off a common and shared set of class libraries, things had gotten a little out of control while Curt was struggling to meet deadlines at Trusty Trucks and Steve working to satisfy Classy Cars. The time had now come to reconcile any differences there might be and to consolidate both versions of the application into a single library.

Hanna knew that once a source file grows bigger than about 400 lines, it becomes hard to read and understand. It was time to break the source into a number of separate files. Each component file should deal with a separate part of the overall system, and, ideally, each part of the system should be dealt with in only one place. The Hacurs team had tried to follow this approach with classic REXX, with limited success. While classic REXX enables the programmer to split procedures off into separate source files, the code in a separate file can see the data passed only as call arguments. This is good in terms of hiding data that the called code should not see or change, but bad when the amount of data that must be shared with the called routine is large. The number of arguments needed on the call statement can become unmanageable, and getting the callers and callee's parameter lists to agree can be difficult.

With Object REXX, sets of related data can and should be grouped together into objects. When a procedure or routine in another file is called, only a short list of objects must be passed. When an object is passed to a subroutine, only a reference, not the object's data, is actually passed. Hanna was trying to work out the best way of applying Object REXX's new capabilities to segmenting the car dealer application code.

Curt and Steve came tramping into the office with a take-out lunch they had bought on the way. They offered Hanna a share.

"Thanks," said Hanna. "How did the training go?"

"No problems," answered Steve. "We agreed to do three more training sessions with them, and that should be all they need from us. How does the converged version of our application look?"

"Just a little chaotic," Hanna answered. "I'm surprised at how many differences you and Curt managed to sneak into your code without consulting anyone. I've drawn up a list of them and have shown what I think the converged version should look like. I'd like you two to go through this list and tell me if you agree or disagree."

Steve and Curt groaned but settled down at their desks and powered up their ThinkPads to review the list that Hanna had stored on the server. About half an hour later, they had completed this task.

"Your suggestions all look good to me, Hanna," said Curt.

"Me, too," agreed Steve. "It's mainly a question of making sure that each class definition includes all the methods and features that we need for both Trusty Trucks and Classy Cars. I think you've got it all sorted out."

"Thanks, guys," said Hanna. "I've also come up with an idea of how we can split the source code into separate files. I'd like to review that with you. We've already moved the major class definitions into separate files. We had to write *::requires* directives in our source files so that the code in each file could see the other classes and methods that it needs." [See *The Requires Directive* on page 76.]

Hanna produced a sketch from her files. "The *Work Order* class requires the *Vehicle* class because it references the vehicle's serial number," she said, "and the *Vehicle* class requires the *Customer* class because it references its owner's ID. The *Work Order* class also requires the *Service* class, since each work order contains one or more services; and the *Service* class references the parts required for a particular service." [See Figure 43.]

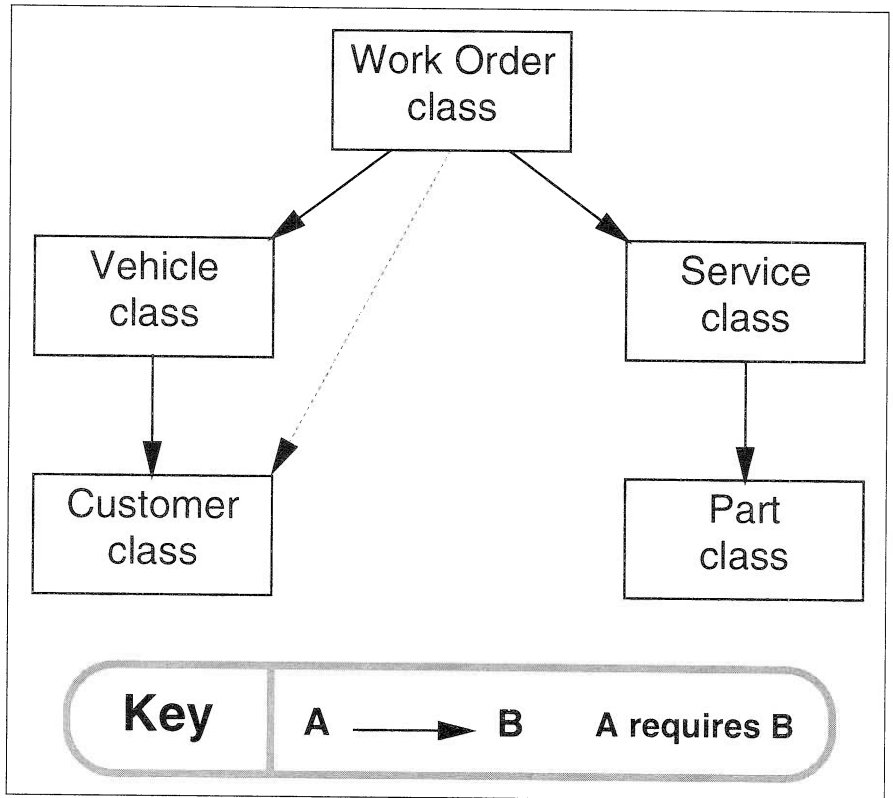


Figure 43. *Car Dealer Data Class Relationships*

“The *Work Order* class also references the *Customer* class,” chimed in Steve.

“That’s right,” agreed Hanna. “I’ll show that with a dotted line. The work order file doesn’t have to contain a *::requires* directive because *Work Order* requires *Vehicle* which in turn requires *Customer*, so the *Customer* class is visible to *Work Order*. Strictly speaking, we don’t need the customer’s ID in the work order object, since we can get it from the vehicle object. But, in theory, a vehicle can change ownership while it’s undergoing service, and there could be arguments about who’s liable for the costs of the service.”

“In practice, the dealer owns the database,” said Steve, “and they would register a change in vehicle ownership only after they had made sure that the new owner would accept the service charges. But Trusty Trucks was worried about this part of the data model, and our zealous salesman bent over backward to meet their needs, as usual.”

“Some of the biggest sales in the history of our industry have been made by salesmen who showed they were keen to meet their customers’ needs,” responded Curt.

Using Multiple Subdirectories

“The way it works out, a lot of the class and method definitions are (or should be) the same across both versions of our application,” she continued. “But some of the method definitions are different, depending on whether it’s the FAT or DB2 version. I wanted to get some uniformity in the file-naming conventions, so I’ve used the same file name for the common or base class definitions, the FAT ones, and the DB2 ones. All *Customer* class definitions are stored in files called `car-cust.cls`, for example.”

“Now hold on, Hanna,” said Curt. “If the base, FAT, and DB2 class definition files all have the same name, they’ll wipe each other out when you copy them into the common directory.”

“I thought of that, Curt,” said Hanna, “and decided that the cleanest approach is to create separate subdirectories for the common base class definitions, and likewise for the FAT and DB2 ones. I store the class definitions each in its own subdirectory, so there are no name conflicts. It could look something like this,” said Hanna, pulling another sketch from her file [Figure 44].

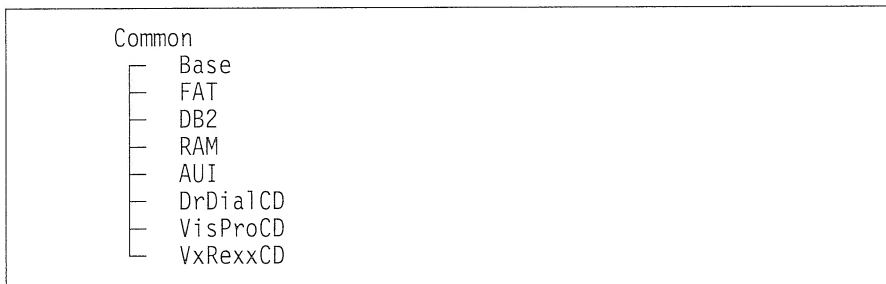


Figure 44. *Directory Structure for Car Dealer Application*

“Why so many subdirectories?” asked Steve.

“I’ve got a list here,” Hanna replied, producing yet another piece of paper:

Common	files common to all configurations
Base	base object management classes
FAT	persistent storage in disk files
DB2	persistent storage in DB2 tables
RAM	initialization of objects in memory
AUI	the ASCII user interface
DrDialCD	the Dr. Dialog GUI builder
VisProCD	the VisPro/REXX GUI builder
VxRexxCD	the Watcom VX•REXX GUI builder

“What’s the RAM subdirectory for?” asked Curt.

“We did our initial development without any persistent storage—remember? I think we can keep that version alive with almost no effort, using the same techniques that we need to separate the FAT version from the DB2 version. I plan to put that code into the RAM subdirectory.”

Controlling Which Files Are Used

“This all looks wonderfully neat and tidy, Hanna,” said Steve, “but how on earth will Object REXX know where to find the files that you’ve hidden in those subdirectories when it runs the application? It will never see them, unless all the subdirectories are in the OS/2 *PATH* environment variable. And if they are, it will always pick the files it needs from the first subdirectory that appears in the *PATH* variable.”

“I thought of that too, Steve,” said Hanna with a smile, “and I built a sample configuration file to try out an idea.” Hanna opened an editor window to reveal the sample code [Figure 45].

```
::requires 'DB2\carcust.cls'  
::requires 'DB2\carvehi.cls'  
::requires 'DB2\carpart.cls'  
::requires 'DB2\carserv.cls'  
::requires 'DB2\carwork.cls'  
  
::requires 'Base\cardeal.cls'
```

Figure 45. DB2 Configuration Command File

“As you can see, I included the relative subdirectory name as part of each *::requires* command,” said Hanna, “and Object REXX was able to find all of the files with no trouble, so long as the common directory was the current directory when I invoked the configuration command file that contains all these statements.”

“That’s pretty neat,” said Steve, “but what happens if the common directory *isn’t* the current directory when you issue the configuration command?”

“It works fine, provided that the common directory is in the OS/2 *PATH* variable,” Hanna responded. “Which is what we would need, even if all our files were merged into a single subdirectory.”

“That sounds great,” chimed in Curt, “but you’ve got the relative subdirectory hard-coded into the configuration command file. What will you do when you need to include the class definition files from the FAT subdirectory instead of DB2?”

“I guess we’ll have to have a different configuration file for each different configuration of files we need to use,” Hanna answered. “It’s sort of like the *make* file you build to tell the C compiler where all the source

files are for a particular project, except the *make* file gets used at compile and link times, while my configuration would be used by Object REXX at run time. In fact, I started out by thinking of all the configurations that we may have to support, and that's what got me to develop a tidy way of handling them all. This is the sketch I made," said Hanna, scratching through her papers. She produced a sketch [see Figure 46].

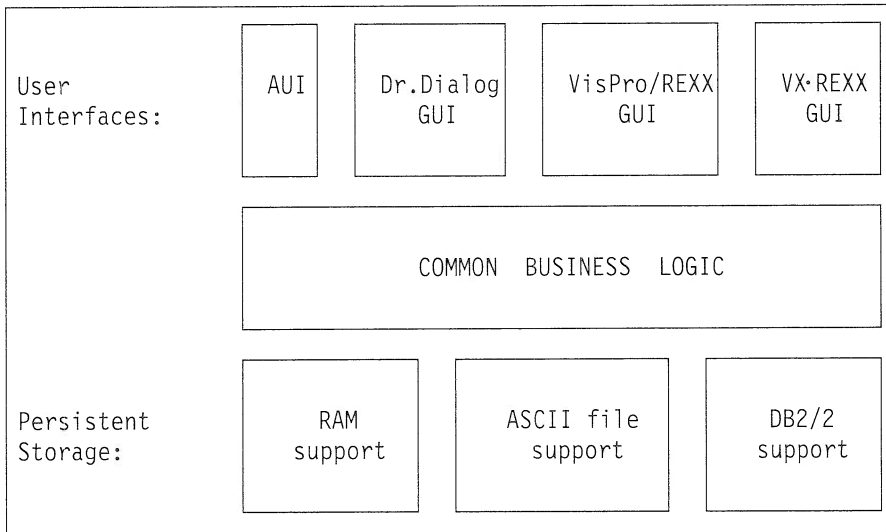


Figure 46. *Car Dealer Application Configurations*

"I've included the storage-based version we started out with for completeness," said Hanna. "I called it the *RAM version*. So, we currently have four different front ends and three different persistent storage systems. In theory, we could support 12 different configurations. And if we succeed in selling our application to other customers, the list of persistent storage systems could grow. We need a way of managing this complexity."

"This is very ingenious," said Steve. "But wouldn't it be simpler just to give every class definition file a different name and put them all into a single, common subdirectory? Suppose you give all the FAT class definitions a file extension of .FAT and all the DB2 definitions a file extension of .DB2. That would resolve the conflict."

"Yes, that would do the trick," said Hanna. "But would it also work when we have to merge the subdirectories that contain the Dr. Dialog, VisPro/REXX, and Watcom VX-REXX projects? We can't necessarily control the names and extensions of all the files those packages produce. And when we come to write the car dealer installation program, I'm sure that it would be easier if all the files we need for DB2 support are in one subdirectory, all the files for FAT in another, and so on.

Then the installation program won't need to know which files are required for each type of support; it will just copy complete subdirectories."

"All this comes from having an obsessively tidy mind," said Curt, "but I can see that it would lead to a tightly controlled system and reduce the number of surprises when we need to implement major new versions of the application. For example, I was talking to an outfit called *Value Vans* the other day, and they are very interested in our application. But they already have several Oracle-based packages running, and we would have to port our code to Oracle before they would even look at it. With this approach, we would create a new subdirectory called *Oracle* and develop the new code we needed in there."

"It also allows us to put fences around portions of the code," said Hanna. "If we get contractors in to develop the Oracle code, for example, we could direct the server to give them read/write access to the Oracle subdirectory and read-only access to the others. That way, they couldn't accidentally break the FAT or DB2 code while they were building the Oracle code."

"That's a good idea," said Steve, "and maybe not just for contractors! I accidentally saved one of my DB2 class definition files on top of Curt's FAT version the other day, and I had to recover his code from the backup tape. I might make fewer mistakes if my default server profile gave me read-only access to the FAT subdirectory. I could always request the server to give me read/write access if I needed it."

"Thanks for your help, guys," said Hanna. "I still need to think this problem through some more. I'll take it home with me. Maybe we can look at it together again tomorrow."

"That's fine by me," said Steve, and Curt grunted agreement, too.

Overall Car Dealer File Structure

The next morning, Hanna was already in when Curt and Steve reached the office.

"Hi, Hanna," they called out as they entered.

"I've got a question," said Steve. "I copied the files you were working with yesterday from the server. When I looked at the end of your configuration file [see Figure 45 on page 148], I noticed a new class file named `cardeal.cls`. What is it?"

"I found that we need a place to put initialization code for all the other classes," Hanna explained. "Every class needs to fetch its initial objects, for example. There didn't seem to be a good place to put it, so I made the car dealer class. It will be responsible for initializing the application and terminating it properly, as well—for example, to disconnect from DB2."

Hanna dug a sketch from her bag. “I was working on the overall structure of our application last night, and it looks like this,” she said [see Figure 47].

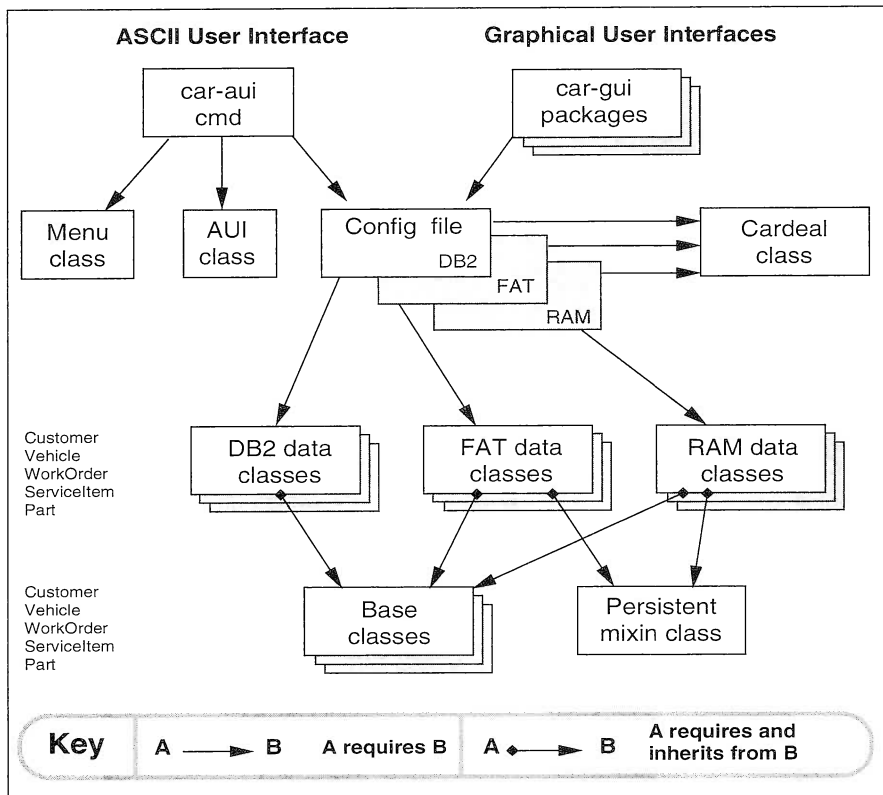


Figure 47. *Car Dealer Application Overall Class Relationships*

“This shows all the files we need for the various configurations we have to support,” Hanna explained. “Each file is shown as a box in the sketch. There were so many, I’ve simplified it by showing boxes stacked on one another. There are *Customer*, *Vehicle*, *Work Order*, *Service*, and *Part* classes all hiding behind the box I labeled *Base classes*, for example, and likewise for the boxes labeled *FAT*, *DB2*, and *RAM data classes*. Each different configuration we need to support will have its own configuration file. I’ve shown them as a stack labeled *Config file*.”

“This looks complicated,” exclaimed Steve. “Do we really need all these files?”

“I think so, Steve,” answered Hanna. “Most of them already exist today for the systems that we’ve developed for *Trusty Trucks* and *Classy Cars*. We just haven’t put all of them together on one piece of paper before. For example, the *car-ai.cmd*, *carai.cls*, and *car-menu.cls* files are all used to drive the AUI interface for the *Trusty*

Trucks version of our application. And the various GUI packages that you and Curt used to build front ends for the system are hiding under the label *car-gui packages*, like spiders under a rock. Those packages actually generate lots of files; I simply haven't shown them."

"Yeah—I guess you're right," said Steve as he stared at the sketch.

"I've shown that there's more than one configuration file," said Hanna. "When a user installs our application onto a PC, the install program will list the various options available, then copy in the configuration file that implements the options chosen. This file will contain *::requires* directives for either the FAT or the DB2 data handling classes. When we need to switch between different configurations, we can edit our own configuration file or copy one of a set of standard configurations into our working directory."

"Now that I can see it all mapped out like this, I realize that we've built quite a complex system," said Steve.

"Have we?" asked Curt. "Or have we built two simple systems and made life difficult for ourselves by trying to share code between them? This looks like a lot of work to me. We aren't a research lab, we're a small software development company. We have to meet customer needs fast, or we'll go out of business. We don't have time to mess around with complicated schemes like this one."

Steve squared up to reply, but before he did so, Hanna intervened by saying "You're absolutely right, Curt. We need to be able to respond to our customers quickly. And we all know that we can't do that with our old invoicing application. We've installed five—no, six different versions of it for different customers. They all started out the same, but today they're all different. Maintaining that code is chewing up a lot of our time. Yet, all of the different versions do much the same thing. We need to be smarter with the car dealer application. You're doing a great job finding prospects for the product. We need to make sure that we can deliver all they need without creating a monster maintenance problem."

Turning back to her sketch, Hanna said "As I said before, most of the files shown in this sketch already exist. We just need to tidy them up so they can reside in the subdirectory structure we looked at yesterday. What do you think, Curt?"

Communication Among Classes

Curt pondered for a while, then said, "Well, for starters, we'll have a problem that one class will not know about other classes. How can a method in the *Customer* class access a method in the *Vehicle* class?" he asked. "Shouldn't every application class have access to all the other application classes, in case we decide to enhance the system?"

Steve had a concerned look on his face, but then he lightened up and shouted, “We could use the Object REXX local directory for this!” He continued, “If every class puts itself into the local directory, all classes will have access to each other.”

The Local Directory

Steve brought up two editor windows with the *Vehicle* and *Customer* classes and changed the source code to make use of the local directory [see Figure 48].

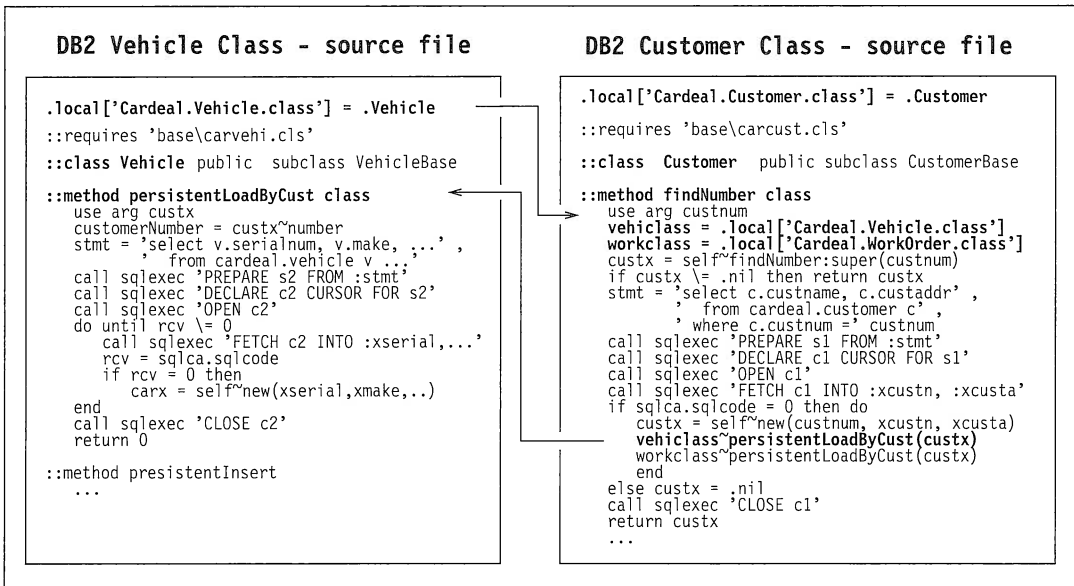


Figure 48. Using the Local Directory

“That’s cool,” said Curt, “I can use the same technique between the *Menu* and the *AUI* class. When the *au*i object is created, I store it in the local directory. This way I don’t need to pass the *au*i object to the *menu* methods.”

Curt thought a few seconds, then said, “Let’s keep our copies of the code that’s currently running at Trusty Trucks and Classy Cars untouched in their respective directories on our server. We can start restructuring the code along these lines, but I would like to be able to show Value Vans that we can meet their requirements as soon as possible. So we can’t afford to get bogged down for weeks in a big restructuring exercise that prevents us from building and running demos.”

“That’s reasonable,” said Hanna. “The question is, how long will it take us to restructure the code along these lines?”

“If we all work at it, I think we could have it done in two or three days,” replied Steve. “Then we’ll have to test it out, of course.”

“Sounds good to me,” said Hanna. “Let’s do it!”

Using the local directory

The local directory object (*.local*) is available to all Object REXX programs running in one OS/2 process.

For the car dealer application we used the local directory to record:

- ❑ Each class as *.local['Cardeal.classname.class']*
- ❑ The relationship between work orders and service items as *.local['Cardeal.WorkServRel']*
- ❑ The ASCII window interface object (*au*) as *.local['Cardeal.aui.object']*
- ❑ The active persistent storage as *.local['Cardeal.Data.Type']*, either FAT, DB2, or RAM
- ❑ The directory of the FAT data files as *.local['Cardeal.Data.dir']*
- ❑ The directory of the multimedia files as *.local['Cardeal.Media.dir']*

The Global Directory

Object REXX also provides a global directory object (*.environment*) that is available to all Object REXX programs running on the same machine. The global directory can be used to communicate between processes, as illustrated in Figure 40 on page 136.

Using the global directory

A small example shows that one process can set up a class and an entry in the global directory, then a second process can use that class through the global directory and, even when the first process ends, the global object will still be there.

eater.cmd (OS/2 window 1)

```
call RxFuncAdd 'SysSleep','RexxUtil','SysSleep'
eater = .eat~new('Peter')
.environment~my.eater = eater      /* global */
say 'The eater is active'
say 'I'm going to sleep now...'
call SysSleep 20                  /* sleep 20 seconds */
.environment~my.eater = .nil
exit

::class eat
::method init
expose name
use arg name
::method feed      /* invoked by other process */
expose name
use arg food
say '===>Eater' name 'got:' food
::method uninit
expose name
say '***> Eater' name 'is terminating!'
```

feeder.cmd (OS/2 window 2)

```
eater = .environment~my.eater
do 10
  say 'Give me a line for the eater'
  say '... or null to end:'
  parse pull line
  if line == '' then exit
  eater~feed(line)
end
exit
```

First start the *eater* in one OS/2 window, then the *feeder* in another OS/2 window.

Installation Program Considerations

Three days later the Hacurs team met to review progress.

“I’ve got my Watcom VX.REXX front end working with the new class structure,” said Curt.

“I’ve got my VisPro/REXX front end working with it, too,” said Steve.

“And I’ve got the old Dr. Dialog front end working with it,” said Hanna with a smile. “Sounds like it’s time for a shoot-out.”

The threesome put their ThinkPads side by side and went through their standard demo process in parallel. Everything seemed to work perfectly.

“Great work, team!” said Hanna. “The class conversion work wasn’t hard at all. Are you running against FAT files or DB2, by the way?”

“FAT files,” said Curt.

“DB2,” said Steve simultaneously.

The two looked at each other. “There’s actually no easy way to tell, short of looking to see which configuration file is currently active,” said Steve.

“I guess that proves something,” said Hanna. “How easy is it to switch from DB2 to FAT?”

“We’ll have to develop an installation program,” said Steve.

“No, it’s simpler than that,” said Curt. He clicked on the directory structure to open the DB2 subdirectory, and drag-copied the configuration file within it back into its parent directory. After confirming the overwrite, he restarted his application. It ran as before, this time using the DB2 configuration file.

“Well that’s a handy way for programmers to do it,” said Steve. “But our users will still need an installation program. Come on, it won’t take long to build. The way Hanna parcelled everything out into separate directories, it should be a snap.”

“Count me out,” said Curt. “I’ve got an appointment to see Value Vans, and I don’t want to be late. See you later.” Curt left with a wave.

“OK,” said Hanna, “I’ll work with you. Should we use a GUI builder tool?”

“Absolutely!” answered Steve. “We want everything about this application to look professional. There, I’ve created a new Dr. Dialog resource file; let’s open it and edit it ... right, I’ve got an empty form. What should I put in it?”

“The target disk and path for our installation,” Hanna answered.

“OK,” said Steve. “I’ll provide an entry field for that.”

“Fine,” said Hanna. “Now we need to offer the user a choice of persistent storage techniques—ASCII disk files or DB2 database.”

“Hmm,” said Steve, “I’ll build a group box labeled *Persistent storage option* and put some radio buttons into it with the two storage options available. I’ll add the RAM option too—let’s call it *Objects in memory*.”

“Now we need to offer the user a choice of user interfaces—the ASCII character, or the Dr. Dialog, VisPro/REXX, or Watcom VX-REXX GUIs,” said Hanna.

“OK,” said Steve, “I’ll copy the storage options group box to make the *User interface option* box. I need an extra radio button, and I must change the text to show the interface options that we have available.”

“And then we need an **OK** button,” said Hanna, as Steve finished this task.

“I’ll put in an **OK** button and a **Cancel** button,” said Steve. “Some folks get a little nervous if they can’t see a **Cancel** button. There, that does it. Now, let’s just neaten this up a bit.” Steve used Dr. Dialog’s group menu buttons to standardize the alignments and sizes of the controls he had built.

“I’ve put hardly any logic behind the controls, but let’s see how it looks,” said Steve. He clicked on the **Tools run** icon and started the application.

“That looks really professional, Steve,” commented Hanna as the window appeared on the screen [see Figure 49].

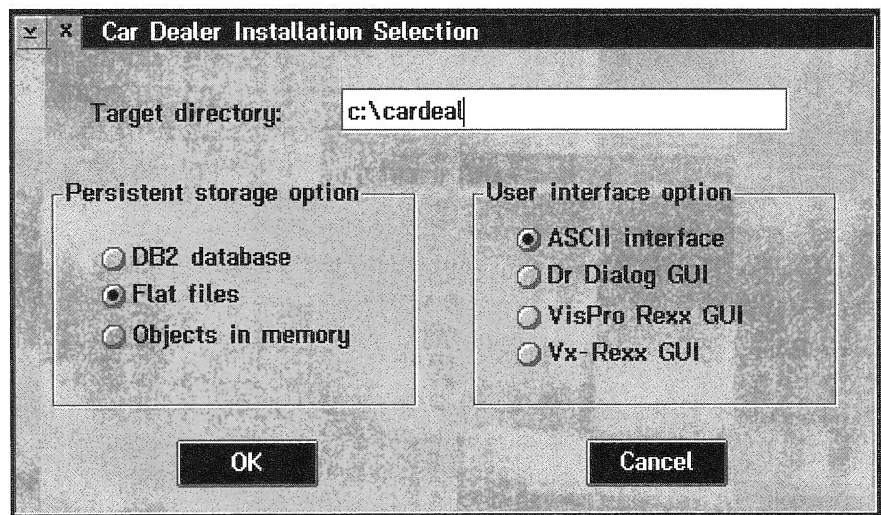


Figure 49. *Simple Car Dealer Installation Program*

“Now I’ve got to add the logic,” said Steve. “But that really shouldn’t be hard, thanks to the directory structure you set up.”

“OK, I’ll leave you to it,” said Hanna, getting up.

“Just a moment, Hanna,” said Steve, rising, too. “There’s something I need to tell you.” Steve suddenly looked serious, and rather strained. “I’ve been thinking about this for a while; maybe it’s time I settled down and sorted out my life. We’ve been so busy getting our company going, I haven’t had time to think about myself. But now that we’ve got our first big application installed with two customers and the money is starting to come in ...” Steve’s voice trailed off.

Hanna felt uneasiness, almost panic. Did Steve want to leave? Their little company had only just started to find its feet, and every member of the team was vital to its continued existence. If Steve left at this stage, Hacurs company would never survive. And what did he mean by “settle down and sort out his life?” Was there a woman in his life? Where had he found time? They had all been so busy getting the company going. Hanna’s heart started to pound.

“So...what I mean is,” Steve struggled on. “Do you want to see the ball game on Saturday?”

A flood of relief swept through Hanna. She laughed involuntarily, and Steve was startled. He might have taken offense, but Hanna’s broad smile and shining eyes reassured him that she would very much like to see the ball game on Saturday.

Implementation of Configuration Files

All configuration files are named `carmodel.cfg`. There is one for FAT persistence, one for DB2 persistence, and one for objects in memory (RAM). Each file is in the subdirectory of its respective implementation.

Object REXX executes any REXX code placed at the beginning of a file required by other programs that use the `::requires` directive. This feature allowed us to place entries in the local directory, load REXX function packages, and then connect to DB2.

Figure 50 shows the configuration file for FAT; Figure 51 shows the configuration file for DB2.

```

Parse source . . me .
maindir = me~left(me~lastpos('\')-1)      /* main cardeal directory */

.local['Cardeal.Data.type'] = 'FAT'          /* Data in Files */
.local['Cardeal.Data.dir'] = maindir'\FAT\Data' /* Data directory */
.local['Cardeal.Media.dir'] = maindir'\Media' /* Media directory */

::requires 'base\cardeal.cls'
::requires 'fat\carcust.cls'
::requires 'fat\carvehi.cls'
::requires 'fat\carpart.cls'
::requires 'fat\carserv.cls'
::requires 'fat\carwork.cls'

```

Figure 50. *Configuration File for FAT Persistence*

```

if RxFuncQuery('SQLDBS') then
  call RxFuncAdd 'SQLDBS', 'SQLAR', 'SQLDBS'
if RxFuncQuery('SQLEXEC') then
  call RxFuncAdd 'SQLEXEC', 'SQLAR', 'SQLEXEC'
call sqlxec "CONNECT RESET"
call sqlxec "CONNECT TO DEALERDB"
if sqlca.sqlcode \= 0 then do; say 'Cannot connect to DEALERDB'
                             exit 16; end

.local['Cardeal.Data.type'] = 'DB2'          /* Data in DB2 */
.local['Cardeal.Data.dir'] = '-none-'        /* Data in DB2 */
.local['Cardeal.Media.dir'] = '-none-'       /* Media in DB2 */

::requires 'base\cardeal.cls'
::requires 'db2\carcust.cls'
::requires 'db2\carvehi.cls'
::requires 'db2\carpart.cls'
::requires 'db2\carserv.cls'
::requires 'db2\carwork.cls'

```

Figure 51. *Configuration File for DB2 Persistence*

Placing the SQL *CONNECT* call into the configuration file completely relieves the ASCII interface and the GUIs from dealing with DB2.

Using the Configuration File

There are two ways of using the configuration file:

- ❑ Put a `::requires` directive at the end of the source program to embed the configuration file:


```

      ::requires carmodel.cfg
      
```
- ❑ Alternatively, *call* the configuration file at the start of the program:


```

      call carmodel.cfg
      
```

The configuration file must either be located in the current directory or be found through the OS/2 *PATH* variable. In our application, we copy one of the three configuration files into the main car dealer directory to make it *active*. Object REXX finds the currently active configuration file—DB2, FAT, or RAM. The application has no knowledge of which persistent storage method was selected. This technique is valid for both the OS/2 window application (AUI) and the three GUI versions. See *How to Include Directives in GUI Builders* on page 92 for detailed instructions.

Configuration File for List Routines

We use an additional configuration file, *carlist.cfg*, to select the correct routines to list customers and work orders for the ASCII user interface according to file or DB2 persistence.

This configuration file is copied from FAT or DB2 subdirectories to the AUI subdirectory automatically, according to the configuration set for persistence.

Implementation of the Car Dealer Class

The *Car Dealer* class is responsible for initialization and termination of the environment. It is also a good place to hold the methods for multimedia—that is, *playaudio* and *playvideo*.

An extract of the class is shown in Figure 52.

```
.local['Cardeal.Cardeal.class'] = .Cardeal

::class Cardeal public

::method initialize class
  self~mciRxInit                                /* initialize multimedia */
  .local['Cardeal.Part.class']~initialize        /* let each class      */
  .local['Cardeal.ServiceItem.class']~initialize /* initialize itself   */
  .local['Cardeal.Customer.class']~initialize   /* and load objects    */
  .local['Cardeal.Vehicle.class']~initialize
  .local['Cardeal.WorkOrder.class']~initialize
  return 0

::method terminate class
  if .local['Cardeal.Data.type'] = 'DB2' then    /* disconnect from DB2 */
    call sqlexec "CONNECT RESET"

::method playaudio class
  ...
::method playvideo class
  ...
::method mciRxInit class private
  ... load multimedia function package
```

Figure 52. *The Car Dealer Class*

Using the Car Dealer Class

Each version of the car dealer application has to make one call, *.Cardeal~initialize*, at the beginning of the program to initialize the application, and one call, *.Cardeal~terminate*, at the end of the program to terminate the application.

Source Code for Configuration Management

The source code for the configuration files is listed in *Configuration for File Storage* on page 310 for persistence in files and in *Configuration for DB2 Storage* on page 317 for persistence in DB2.

The source code for the *Car Dealer* class is described in Table 21 on page 255 and listed in *Cardeal Class* on page 308.

11

Object REXX, SOM, and Workplace Shell

In this chapter, we extend the car dealer application, using some of the facilities that Object REXX provides for interacting with SOM. SOM is a standard component of OS/2, AIX, MVS, and OS/400 that enables objects to interact with other objects by exchanging messages with them, even when the other objects are parts of different applications, written in different languages and running on different, distributed computers. It provides a very powerful but easy-to-use way of developing client/server applications.

SOM conforms to the CORBA standard, by far the most advanced and widely implemented standard for interobject communication available today. Objects that can communicate with SOM can also interact with objects running under any other CORBA-compliant broker.

In this chapter, we also make use of Object REXX's SOM facilities to access the OS/2 WPS.

Using SOM in the Car Dealer Application

Curt came clattering into the office, wielding a huge umbrella that was shedding water copiously. “Wow! It’s raining cats and dogs out there,” he said, laying down the umbrella and brushing water from his jacket. “Hi, Hanna,” he added, “Where’s Steve?”

“Hi Curt,” Hanna responded. “You’re right about the rain. I’ve been watching people struggle past my window. Just about 20 minutes ago the gutters were so full, they ran over onto the sidewalks. Steve’s still busy with Classy Cars. They’re working out an implementation schedule for centralizing their data onto a single, secure server. How did your call on Value Vans go, by the way?”

“Very well,” said Curt. “They loved the demonstration, and we went through all their requirements this morning. Our application can meet almost all of them as it stands.”

“That sounds encouraging!” said Hanna. “What sort of things do they need beyond what we can currently do?”

“They want to do a lot more in the area of accounts analysis,” said Curt. “I told them that if they store their data in DB2, they’ll be able to access it using any of a number of shrink-wrap PC packages with analysis facilities. Since DB2 provides support for the ODBC API and since most PC data analysis packages can make use of it, we have lots of choice in this area. They make extensive use of Lotus 1-2-3 already, so I thought we could help them access their data in DB2 from Lotus 1-2-3. If we set up the ODBC interfaces for them and code a few sample spreadsheets and macros, we can get them off to a flying start.”

“Good thinking, Curt,” said Hanna. “Maybe we should code some examples and store them on our ThinkPads. That way, we could show our prospects how our approach would work.”

“You’re right, of course,” Curt agreed. “I’ll try my hand at doing that. Oh, and there was one other thing that Value Vans asked about. They’re in the process of installing a client/server stores management system. It’ll help them manage stock levels and procurement. This system will manage the parts they use when they service vehicles, so they want us to interface our car dealer application to it. We’ll have to get all our parts information from this system.”

“Hmm—I wonder how easy that will be,” said Hanna. “Will the stores system be running on the same processor as our application?”

“The client code will run on the same PCs as do our application’s clients,” Curt replied. “But their server code needs to run on an AIX machine.”

“Oops!” said Hanna, “Our server code was specifically designed to run under OS/2, so we won’t be able to run it on the stores server. What database manager does the stores system use? Maybe we can use remote data access to get hold of the parts.”

“The stores system uses flat files,” Curt replied. “But don’t worry. I found out that this system is being developed in C++, and the clients will make use of SOM to communicate with the server. Since Object REXX allows access to SOM objects, I’m sure we can adapt our system to work against their parts.”

“I like your confident approach, Curt,” said Hanna. “I’m sure that’s why you’ve been so successful in selling our products. But we’ll have to research this situation quite carefully. At this stage, Object REXX allows us to access SOM objects built in other languages, but we can’t create SOM objects in Object REXX.”

“Well, I happen to know that both AIX and OS/2 support SOM objects and that they can communicate with one another,” said Curt. “The stores system implements parts, along with other stock items, as a SOM object, so we should be able to get hold of it from Object REXX.”

“That’s true, Curt,” Hanna agreed, “but we need to know more about the way they’ve implemented their stock item. Does it contain all the information our application needs, for example? Tell you what, if you can get the IDL—that’s Interface Definition Language—that defines their stock item and its behavior, then we can work out how difficult it would be to use it. This may not be quite so easy as the other installations we’ve done.”

“OK,” Curt agreed, “I’ll get a copy of the IDL for their stock items. Is there anything else we need?”

“Well,” Hanna replied, “we will need to get access to their stores server from our development machines here so we can make use of their stock-item SOM object.”

“I’ll check it out,” said Curt, “I’m sure there won’t be a problem.”

Hacurs Builds a SOM Object

The next morning, when Curt came in, both Hanna and Steve were in the office.

“Hi guys,” Curt called out, “I’ve got the IDL for the stock-item SOM object. It’s only a small file; I’m sure this is going to be easy.” Curt unpacked his ThinkPad and plugged it in. While it powered up, he continued. “I asked about getting access to their stores server, and there’s no problem. We can use a SLIP connection to a dial port on their server. We can use the TCP/IP that ships as a standard component of the Warp Bonuspack or Warp Connect.”

“Well, there’s a bit more to it than that,” said Hanna. “Thanks for bringing in the IDL, Curt, but I’m quite worried about this project. None of us has used SOM or distributed SOM [DSOM] before, and we’re bound to hit some problems. And learning how to use DSOM

with a SLIP connection to a remote server that we can't see—it's going to be very difficult to know what's going wrong where, when things don't work."

"Come on, Hanna!" said Curt in exasperation. "We're all ace programmers, aren't we? DSOM is supposed to make it very easy to implement distributed objects. Sure, we'll have to learn things as we go along, but we've done that before."

"Yes," Steve agreed, "but we'll be going through the DSOM learning curve on your customer's server. They'll be able to see every mistake we make. We could blow our credibility before we're able to finish implementing the system and show them how well it works."

"Oh—well, maybe we can get their stores system to run on our OS/2 server here," suggested Curt. "Then we can cut our teeth on SOM while no one's looking over our shoulder."

Hanna and Steve looked at each other. "That's a good idea, Curt, but I doubt if their stores system will run under OS/2. Since they've coded it in C++, they've probably made direct calls to AIX for the resources they need. And the AIX APIs are different from the OS/2 ones."

"But this is C++ code, Hanna," said Curt. "C++ comes with a very comprehensive class library. I'm sure that it provides all the resource management their programs need, so they won't have to use AIX APIs. And the C++ class library is compatible between AIX and OS/2. That means their code should be portable, too."

"Maybe so," said Steve, "but I wouldn't count on it. There are a lot of unreconstructed C programmers out there. And, anyhow," he continued, "would they be prepared to give us their source code? They probably regard us as competitors."

"Well, there's no point in us sitting around here trying to second-guess what they have done and what they will do," said Curt. "I've met with the suppliers of the stores system, and I've got their phone number. I'll call them and ask if we can have their source code for the stock item and whether we can port it to OS/2."

While Curt made the call, Steve had a look at the stock-item IDL file. Curt's conversation was rather short, and he was scowling when he put the phone down.

"The project leader thinks they have made some use of AIX APIs," he said. "And he says it isn't possible to separate out just the stock-item SOM object and run it on its own. It interacts with all the other SOM objects in their system."

"Well, is he prepared to give us the source for the whole system?" asked Hanna.

"Not at this stage," Curt answered. "Seems like their system is only 90% complete, and they aren't prepared to start handing it out yet."

"Uh, oh!" said Hanna.

“What’s with the ‘Uh, oh,’ Hanna?” asked Steve. “Do you mean OO?”

“Uh-uh!” said Hanna, shaking her head in disagreement. “Not this time. This is the other ‘Uh, oh’ that’s been around the IT industry a lot longer—the one no one likes to talk about. You used the ‘90% complete’ phrase. In the company we used to work for, that meant the project was in trouble. Tell me Curt, when is the stock system due to be installed?”

“Four weeks from now,” Curt replied.

“And when would they install our car dealer system, if they went for it?” asked Hanna.

“Only afterward,” Curt replied. “They don’t want to install two systems at the same time—it would be too disruptive.”

“That makes sense,” said Hanna. “But what will happen if the stores system goes in late?”

“Then our implementation would slip, too,” Curt answered.

“If the stores system does go in late, would they consider implementing our car dealer system as is, and changing it later to work with the stores system when it gets implemented?” asked Steve.

“Now, hold on!” said Curt. “Value Vans has had a lot of its people working on the stores system for the last three months. I’ve been talking to them for only about one month. If I suggest to them that they’re going to be late and that we’ll be ready before they are, they’ll throw me out of their offices.”

“I guess you’re right,” said Hanna, “but I think we need to keep a fall-back implementation plan up our sleeves, just in case the stores system isn’t there when we need it. We’re getting good revenue from Trusty Trucks and Classy Cars, but if we get stuck with a two-month or longer holdup at Value Vans, through no fault of our own ...”

There was a glum silence while the Hacurs team contemplated this possibility. Then Steve piped up, “I’ve got an idea. I’ve had a quick look at the IDL for the stock item, and it looks pretty simple. Even if the suppliers don’t give us the source, I reckon that I could knock together a SOM object in C++ that implements the behavior we need pretty quickly.”

“Really, Steve?” asked Hanna. “Let’s have a look at the IDL.” They all turned to look.

“We don’t need many instance methods for the part object,” said Steve.

“That’s true,” agreed Curt, “and the ones we do need are pretty simple.”

“But what about the persistent storage methods?” asked Hanna. “I don’t see any in the IDL. Maybe they’re hidden inside the *create*, *update*, and *delete* instance methods?”

“I guess they must be,” Steve agreed.

“Well how would you handle it?” Hanna asked. “Do you want to code persistence methods in C++? And which would you code—the FAT ones or the DB2 ones?”

Steve shook his head. “I don’t want to code either of those in C++,” he answered. “Hmm. Maybe we can subclass the SOM objects class in Object REXX and add our existing persistent storage methods to the new subclass. Then the rest of our car dealer system can use the Object REXX subclass for parts, and I won’t have to code the persistence methods in C++.”

“Have you got time to build it, Steve?” asked Hanna.

“Yes,” Steve answered. “The plans I worked out with Classy Cars leave me a few days free now. I can build the C++ code, but you or Curt will have to write the Object REXX code.”

A slow smile crept across Hanna’s face. “Don’t you love it when a plan comes together?” she said. “I think we can make this work. If Steve builds an OS/2 version of the stock-item SOM object, we can do our car dealer customization and testing in our own office and implement at Value Vans only when our code is solid. If the stores system goes in on time, we use it. Our code will be enabled to work with a stock-item SOM object already, so it shouldn’t take long to sort out the connection. And if the stores system is late, we use our own stock-item SOM object as a stopgap measure, and we’re still able to implement on schedule. Either way, we come out smelling like roses.”

Steve had risen to go to his desk. As he passed behind Hanna, he sniffed loudly and appreciatively at her hair. “Hmm, I love that rose scent!” he said.

Hanna crumpled up a piece of paper and threw it at him, but he ducked out of the way and went to his desk to start building the stock-item SOM object.

How the SOM Object Was Implemented

Two days later, Steve announced that his stock-item SOM object was ready for testing. Hanna and Curt pulled their chairs around his desk to see.

“Let’s have a look at the IDL you built for your stock item first,” said Hanna.

Steve brought up his IDL in an editor window. “I called it *Part* rather than *Stock Item*, but that’s no problem, it’s easy to change,” Steve said. “Now don’t get picky about this, it’s throwaway code, remember. It’s here to help us do our own development and testing, and maybe as a stopgap solution if the stores system is late, but after that we don’t need it. So I made things as easy as possible for myself.”

Hanna and Curt said nothing, but exchanged smiles. Steve was a perfectionist and embarrassed to show them code that was anything less than a masterpiece.

“We can’t define both class and instance methods in a single SOM class,” Steve continued, “so I’ve defined two separate but related SOM classes. I built a SOM Object called *Part* for the instance methods, and a SOM Class called *PartMeta* for the class methods. I told SOM to give *Part* the class methods from *PartMeta*. Let’s look at *Part* first.” [See Figure 53.]

```
#include <somobj.idl>
#include <somcls.idl>
#include "partmeta.idl"
interface Part : SOMObject
{
    attribute short pid;           // part number
    attribute short pprice;        // part price
    attribute short pstock;        // part stock
    attribute string pdesc;        // part description
    short number();                // get number
    short price();                 // get price
    short stock();                 // get stock
    string description();          // get description
    string detail();               // make a detail line
    void display();                // display to standard out
#ifdef __SOMIDL__
    implementation {
        releaseorder: _get_pid, _set_pid, _get_pprice, _set_pprice,
                     _get_pstock, _set_pstock, _get_pdesc, _set_pdesc,
                     number, price, stock, description, detail, display;

        metaclass = PartMeta;
        majorversion = 0;
        minorversion = 0;
        dllname      = "part.dll";
        // # Method Modifiers
        somInit: override;
        somUninit: override;
    };
#endif /* __SOMIDL__ */
};
```

Figure 53. IDL for the SOM Object *Part*

“Let me walk you through this,” said Steve. “From the top,

- ❶ “I start with some standard includes for the SOM header files.
- ❷ “I include the *PartMeta* IDL as well. We’ll look at that later.
- ❸ “This is where the interface definition starts. I tell SOM that *Part* inherits from *SOMObject*, the root of all SOM classes.
- ❹ “The attribute tags identify instance variables and also automatically create *get* and *set* methods for each. Since it’s C++, these methods start with an underscore character. So the methods for *pprice* are *_get_pprice* and *_set_pprice*, for example. We won’t be

using these methods because their names are different from those we've already coded in Object REXX. Notice that I had to put an extra *p* in front of each instance variable. SOM doesn't allow me to have an attribute name that is the same as a method name.

- 5 "This is where I define the methods we use in our Object REXX code.
- 6 "This is where the *Part*'s implementation definition starts. I have to specify the order in which its methods are released.
- 7 "Here's where I tell SOM that *PartMeta* is *Part*'s meta class. *Part* gets *PartMeta*'s methods as its class methods.
- 8 "My compiled and linked code is stored in *part.dll*.
- 9 "I tell SOM that we need to override the SOM *init* and *uninit* methods so we can keep track of our objects in persistent storage."

Steve continued, "Now let's look at the *PartMeta* IDL." [Figure 54].

```

#include <somobj.idl>
#include <somcls.idl>
    interface Part;           // forward declare
    interface PartMeta : SOMClass
    {
        attribute      sequence<Part>  pextent;    // extent of instances
        void            add(in Part partx);        // add part to extent
        void            remove(in Part partx);     // remove part from ...
        sequence<Part>  extent();                 // retrieve extent
        Part            findNumber(in short pnun); // find part by number
        string          heading();                // list heading
#ifdef __SOMIDL__
        implementation {
            releaseorder: _get_pextent, _set_pextent,
                          add, remove, extent, findNumber, heading;
            majorversion = 0;
            minorversion = 0;
            dllname       = "part.dll";
            /*# Method Modifiers
            somInit: override;
            somUninit: override;
            */;
        };
#endif /* __SOMIDL__ */
};

```

Figure 54. IDL for the SOM Class *PartMeta*

"From the top,

- 1 "I start with the same standard SOM includes.
- 2 "*PartMeta* refers to *Part*, so I need a forward declaration.
- 3 "Here's the interface definition. I tell SOM that *PartMeta* inherits from *SOMClass*, so its methods will be class methods. *Part* has only one class attribute, and that's the extent. I implement that as

a sequence of *Parts* in my SOM code. Then I identify the class methods our Object REXX code needs: *add*, *remove*, *extent*, *find-Number*, and *heading*.

4 “Here’s the implementation definition, where I specify the order in which its methods are released.

5 “I tell SOM that we need to override the SOM *init* and *uninit* class methods.”

“So that’s the IDL you had to write,” said Curt thoughtfully. “There isn’t a lot of it, but—rather you than me!”

“Rather the compiler than me, next time,” said Steve with a smile. “The VisualAge C++ product compiler can generate IDL automatically from the class and method definitions you write in the C++ code.”

“This is great, Steve,” said Hanna. “What do we need to code in Object REXX to make use of this?”

Steve looked sheepish. “Hey, I couldn’t give you this code without testing it, could I?” he asked. “So I just wrote a little Object REXX code to make sure it works OK. Here it is,” he said, opening a new editor window [Figure 55].

```

::class SOMPart    EXTERNAL 'SOM Part'           1

::class PartBase public subclass SOMPart         2

::method initialize class                         3
    self~persistentLoad

::method init                                     4
    use arg partid, description, price, stock
    self~_set_pid(partid)
    self~_set_pprice(price)
    self~_set_pstock(stock)
    self~_set_pdsc(description)
    self~class~add(self)
    if arg() = 5 then self~persistentInsert

::method increaseStock                           4
    parse arg stockchange
    self~_set_pstock(self~stock + stockchange)
    return self~persistentUpdate

...

```

Figure 55. *Object REXX PartBase Class for SOM*
An extract of the beginning of the file is shown.

“I took the base class definition for *Part* and modified it to work against my SOM *Part* object,” Steve continued. “I won’t go through all the detailed changes, but I want to point out a few things:

- 1 “This is how we import the SOM *Part* class and its associated meta class, *PartMeta*. Object REXX reads the SOM interface repository and builds an Object REXX class called *SOMPart*, which gets all the instance methods of the *Part* SOM object and all the class methods of the *PartMeta* SOM Class.
- 2 “I could have defined *PartBase* on the first line, but we need to add some extra class and instance methods that we have already coded in Object REXX. So instead I defined *PartBase* as a subclass of *SOMPart*. It inherits all of *SOMPart*’s class and instance methods.
- 3 “The class and instance method definitions that we need to add follow immediately after Step 2, as usual. I copied them from the original *PartBase* and modified them as needed to invoke the SOM *set* methods.
- 4 “For example, I coded the *init* and *increaseStock* methods.”

“This looks really simple, Steve,” said Hanna. “I was afraid we’d get bogged down in a long coding exercise.”

“Object REXX’s support for SOM makes it simple,” Steve replied. “Mind you, I haven’t put in all the changes that will be required to the base class to support the Part SOM object. I thought you and Curt might like to do that. It’s mainly a case of using the *_set* and *_get* methods that SOM generated to set and get the part’s attributes.”

“This is great, Steve,” said Curt. “You’ve got us off to a flying start. It shouldn’t be too hard to finish the job.”

Implementation Steps

A number of steps are needed to make the SOM Object *Part* operational from IDL. The CSet++ compiler and the SOM Toolkit are prerequisites. [See Figure 56 on page 171.]

- 1 Check the *SET SOMIR* statement in CONFIG.SYS. Any SOM compilers use the last SOM interface repository in that concatenation. It may be desirable to define a separate SOM.IR file in the car dealer directory.

- 2 Run the SOM compiler against *part.idl* and *partmeta.idl*:

```
/* config.sys overwrites */
/* pointer notation      */
SET SMADDSTAR=1          /* c++ and def emitters */

SC.EXE -u part.idl        /* compile the IDL files */
SC.EXE -u partmeta.idl
```

This generates *.xh*, *.xih*, *.cpp*, and *.def* files.

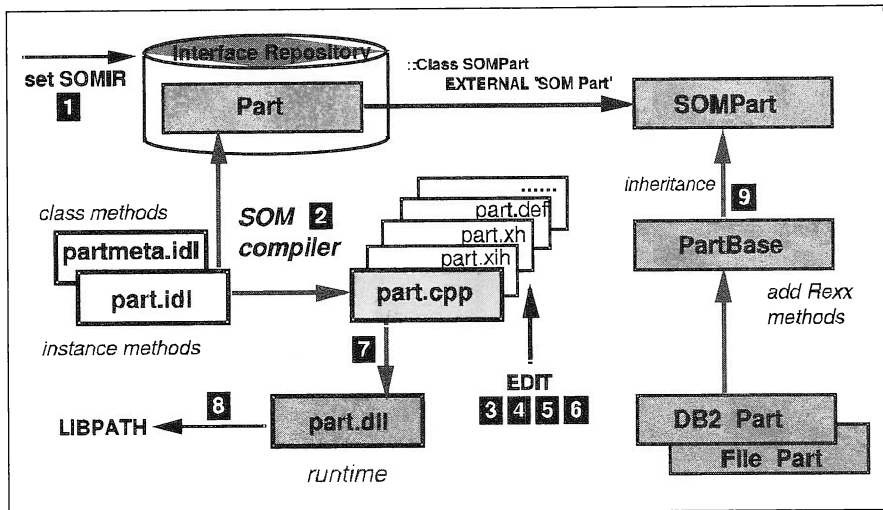


Figure 56. Implementation Steps for SOM Object Part

- 3 Replace the `_set_pdesc` method in `part.xih`. The SOM-generated code does not handle strings properly.

```
SOM_Scope void SOMLINK _set_pdesc(Part *somSelf, Environment *ev,
    string pdesc){
    PartData *somThis = PartGetData(somSelf);
    PartMethodDebug("Part", "_set_pdesc");
    SOM_IgnoreWarning(ev);
    /* somThis->pdesc = pdesc; ***** eliminated *****/
    /****** replacement *****/
    if (somThis->pdesc)
        SOMFree(somThis->pdesc);
    somThis->pdesc = (string) SOMMalloc(strlen(pdesc) + 1);
    strcpy(somThis->pdesc, pdesc);
}
```

- 4 Add method code to `part.cpp` for the *number*, *price*, *stock*, *description*, *detail*, *display*, *somInit*, and *somUninit* methods.
- 5 Add method code to `partmeta.cpp` for the *add*, *remove*, *extent*, *find-Number*, *heading*, *somInit*, and *somUninit* methods.
- 6 Combine `part.def` and `partmeta.def` into one `parttot.def` by concatenating the export entries.
- 7 Compile `part.cpp` and `partmeta.cpp`, and link them into `part.dll`:

```
ICC -I. -I%SOMBASE%\include -Q+ -W3 -Gd- -Ge- -c part.cpp
ICC -I. -I%SOMBASE%\include -Q+ -W3 -Gd- -Ge- -c partmeta.cpp
ILINK /packd /packc /exepack /align:16 /noi /noi /De /PM:V10 /Freeformat
    part.obj partmeta.obj /OUT:part.dll somtk.lib os2386.lib parttot.def
```

- 8 Copy `part.dll` into a `LIBPATH` directory.
- 9 Implement the Object REXX *PartBase* class, as shown in Figure 55 on page 169.

Running the Application with the SOM Part

The prerequisites for running the car dealer application with the SOM *Part* class are:

- ☐ Copy the `part.dll` file into a *LIBPATH* directory.
- ☐ Make sure that the `SOM.IR` file of the car dealer directory is in the concatenation in *SET SOMIR* in `CONFIG.SYS`.
- ☐ Replace `carpart.cls` in the Base subdirectory with the SOM version of the *Part* class (`part.som`).

Implementation Notes

1. We added our own get methods to the SOM IDL to make it consistent with our Object REXX class. These methods are duplicates of the SOM-generated *_get* methods.
2. We implemented persistence outside the SOM class to enable persistence in files or DB2.
3. We used a SOM sequence attribute in `partmeta.idl` to keep track of up to 30 part objects (in `partmeta.cpp`).

Source Code for SOM Implementation

The source code for the SOM implementation is not listed in the appendix; it is available in the car dealer directory on the CD or after the sample applications have been installed on a hard drive (see Table 30 on page 258).

Object REXX and the OS/2 Workplace Shell

One morning, about a week later, Hanna was already in the office when Curt and Steve came in. She looked up from her work and greeted them. “How’s Classy Cars doing, Steve?” she asked.

“Very well,” Steve replied, “and I’ve got a lot of the development done in preparation for their centralization in a month’s time. How are things progressing at Value Vans, Curt?”

“They’re getting more and more worried about getting their stores system in on schedule,” Curt answered. “On the other hand, they’re really delighted with the demo I gave them of our car dealer application running against a SOM Parts object. I’ve started talking to them about contingency plans to get our car dealer system in on schedule if stores happens to be late, and they’re listening.”

“That sounds great, Curt,” said Hanna.

“And what have you been doing back in the office while we’ve been slaving away with our customers, Hanna?” asked Curt.

“I’ve been looking at Object REXX’s new WPS facilities,” Hanna answered. “I’ve finished all the coding you two asked me to do,” she continued a little defensively, “so I felt I could invest some time in doing a little research.”

“Hmm—research, hey?” said Curt with a solemn look. “I wonder if we could apply to the government for a research grant...”

“Oh don’t be silly!” said Hanna, slightly flustered. “Anyhow, look at what I’ve got going here. It might come in handy one day.”

Steve and Curt drew their chairs up to her desk.

“Classic REXX has always had functions that allow REXX commands to access and manipulate the WPS,” said Hanna. “Object REXX has the same functions,” she continued, “but also offers the programmer more direct and powerful means of doing this job. The OS/2 WPS is an OO system in its own right, and it’s SOM-enabled. Object REXX’s SOM support can access and manipulate WPS objects directly. Any WPS class may be imported into Object REXX in exactly the same way as any other SOM class.” [These new facilities are described in the publications listed in *Related Publications* on page xxvii, and there are some good examples of their use in the Object REXX samples subdirectory.]

Car Dealer Data in the Workplace Shell

"I cannibalized a copy of the `car-auri.cmd` to build this little demo," said Hanna.

"The new command is called `carshow.cmd`. It invokes the `initialize` method of the *Car Dealer* class to get all of our car dealer objects loaded into storage, then displays them and their relationships in a WPS folder. I put my command in a subdirectory called *WPS*, so if I just type in `wps\carshow` at the command line—there we go."

Hanna's new command created a new icon called *Car Dealer Show* on the desktop and opened it to reveal a folder with a **Customer View** icon and several template icons inside it [see Figure 57].

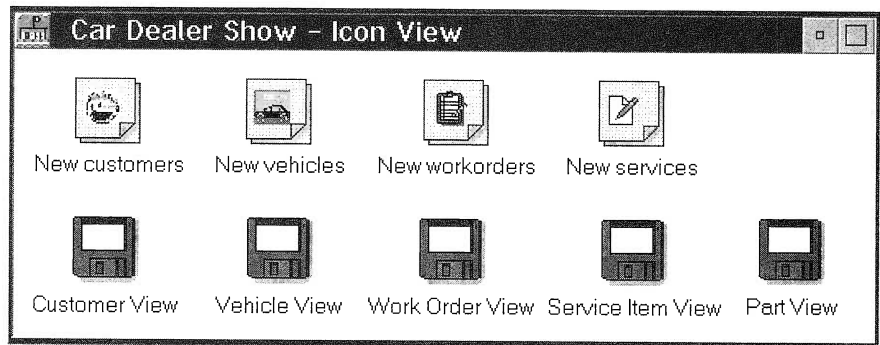


Figure 57. *Car Dealer Show WPS Folder*

Hanna's PC cranked away for a little while longer.

"What's happening, Hanna?" asked Steve.

"It's populating the *Customer View* folder, but I'll show you in a minute," Hanna replied. "Let's just have a look at the rest of the folder that my command opened. There are templates for new customers, vehicles, work orders and service items. Now let's look at the *Customer View*." Hanna selected the *Customer View* folder. It was populated with an icon for each of the customers in the sample database, each tagged with the customer's number and name [see Figure 58].

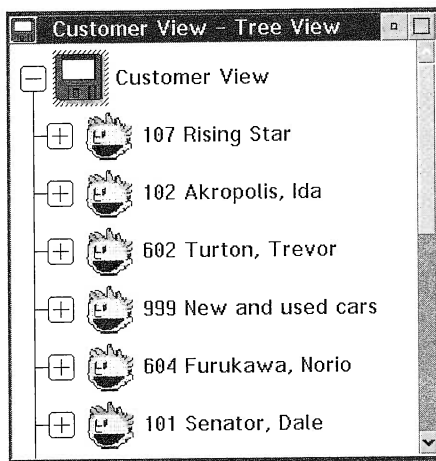


Figure 58. *Car Dealer Customer View Folder*

“Hey, that’s cute!” said Steve.

“It gets cuter,” said Hanna. She clicked on the plus sign in front of *Rising Star*, and the folder expanded to show an icon for Rising Star’s car—an *Acura-Legend*. [See Figure 59 for this and the subsequent expansion steps.]

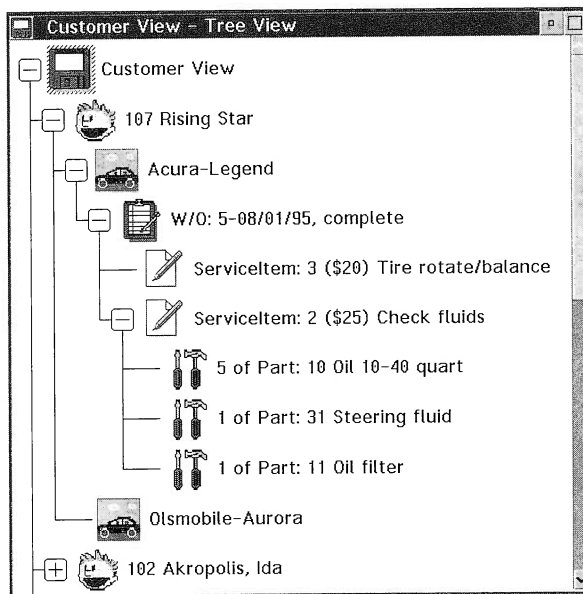


Figure 59. *Car Dealer Customer View Folder, Expanded*

“Neat,” said Curt.

“It gets neater,” said Hanna. She clicked on the plus sign in front of the **Car** icon, and a **Work Order** icon appeared below it. She expanded the work order, and two **Service Item** icons appeared. She clicked on a service item, and three **Part** icons filled in below it.

“Wow!” said Steve and Curt.

Hanna then collapsed the complete *Rising Star* branch by clicking on the minus sign before the customer’s icon. Next, she expanded a few other customer icons. “As you can see, the whole sample database has been expanded into this folder in the form of icons,” Hanna said. “This representation could be used to give a customer-centered view of the data to the car dealer staff that deals with customers. The program goes on and builds similar representations that are rooted on vehicles, work orders, service items, or even parts.” [See Figure 60.]

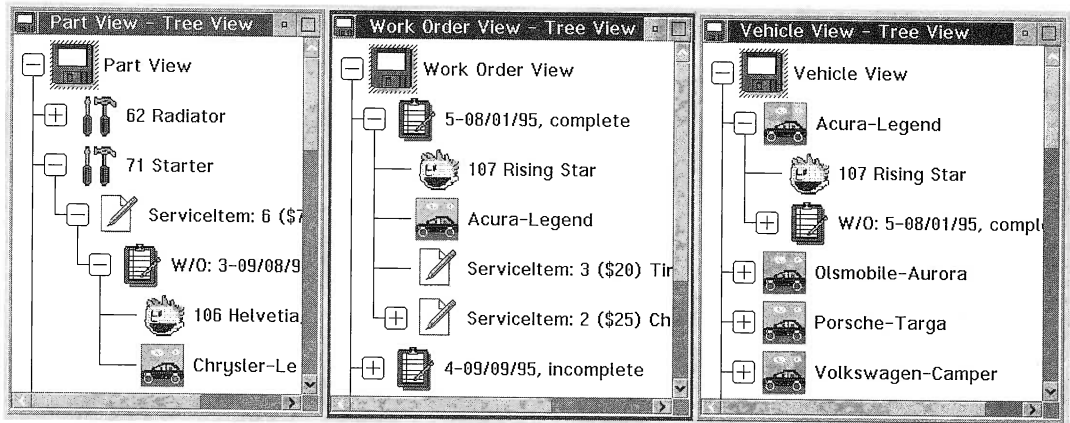


Figure 60. Car Dealer Views

“That’s a pretty compact way of putting a lot of data up on the screen,” Curt mused. “The user can drill down into any area of interest.”

“What about the first folder you opened, Hanna—the one with the templates in it?” Steve asked.

Hanna smiled. “So far, all I’ve shown you is a way to present information to the user,” she said. “I was wondering if this approach could be extended to allow the user to manipulate data. So I defined these templates. Let me show you what I can do with them.”

Hanna dragged the *New vehicles* template and dropped it onto the **Acropolis, Ida** customer icon. A new car joined the other two below. Hanna then dragged the *New workorders* template, dropped it onto the vehicle, then dropped a *New services* template onto the work order [see Figure 61].

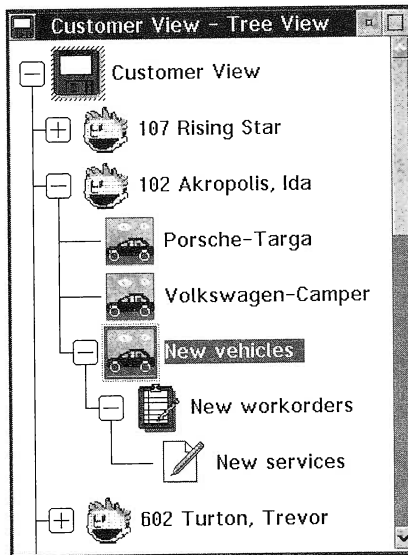


Figure 61. *Customer View Folder Populated by Drag and Drop*

“What I was playing with here is a way of allowing the user to update the system using mainly drag and drop,” said Hanna. “It’s a very simple and intuitive interface.”

“This all looks amazing, Hanna,” said Steve, “but how do you specify the details of the customers, cars, and services that you drop? So far, you’ve left them all with default values.”

“At this stage, I can’t handle updates, only data presentation,” Hanna answered. “I can import all the SOM classes that define the WPS into my application, and invoke their methods to create new WPS objects and manipulate them. But I don’t get any notification from WPS when the user manipulates the icons I create.”

“Is there any way of getting that feedback?” Curt asked.

“Not as Object REXX stands today,” Hanna answered. “I can’t subclass the WPS classes that I import. If this were possible, I could write methods for my subclasses that would trap the WPS messages associated with opening, dragging, and dropping WPS objects. So, if the user dropped a *New car* onto a customer, I could open up a GUI panel to allow the user to capture the new vehicle’s details. I would, of course, forward the drop message to my superclass, which would invoke standard WPS processing.”

“What happens if you do stupid drags and drops?” Steve asked. “Say, for example, you pick up an existing customer in your database view and drop it onto an existing service item?”

“The WPS will do exactly as I tell it to do,” Hanna answered. “If I could subclass the WPS class and write my own drag and drop methods for the objects that I put up on the screen, I could decide which drop targets were meaningful and outlaw all the rest. The user would get the normal no-entry sign unless the icon were hovering over a legal drop target.”

Implementation Notes

1. The *WPSINST* command must have been run in the Object REXX directory.
2. First steps in the program are to initialize the *Cardeal* class, import the *wpAbstract* class, and call the *wpconst* command provided by Object REXX.

```
.Cardeal~initialize
wpAbstract = .wps~import('WPAbstract')
call wpconst /* define WorkPlace Shell constants */
```

3. The program basically reads through all of the objects and creates a folder for each one encountered. The basic Object REXX code to create the *Customer View* folder is very simple:

```
dealer = addFolder('Car Dealer Show', dealicon, .wpdesktop)
custView = addFolder('Customer View', dataicon, dealer)
do custn over .Customer~findName('') /* get all customers */
  customer = .Customer~findNumber(custn~left(3))
  custFolder = addFolder(customer~makestring~substr(11), ,
                        custicon, custView)
  do car over customer~getVehicles
    carFolder = addFolder(car~makemodel, caricon, custFolder)
    ...
    ... work orders, service items, parts ...
    ...
  end
end

addFolder: procedure
  use arg name, iconFile, parent
  folder = .wpfolder~new(name~makestring, '', parent, 1)
  folder~wpsetup('NODELETE=NO;ICONFILE='iconFile)
  folder~wpSetDefaultView(.wpconst[OPEN_TREE])
  return folder
```

Source Code

The source code for the WPS demonstration is not listed in the appendix; it is available in the car dealer directory on the CD or after the sample applications have been installed on a hard drive (see Table 32 on page 258).

Applications Assembled from Components

Steve and Curt looked at Hanna's little demo thoughtfully. With only about 200 lines of code, Hanna had put a whole new slant on the car dealer application.

"I wonder if this is the way we'll be developing applications in the future?" Steve mused. "It strikes me that a lot of the objects our programs deal with could be presented in this way. It would give the users a simple and uniform way of seeing and changing the relationships between different objects. By the way, why did you use the WPS to display application data?"

"Ah—I guess I could quote Sir Edmund Hillary," said Hanna. "When asked why he had climbed Mount Everest, he said, 'Because it's there!' I wanted to add some new function to the car dealer installation program we developed," Hanna answered, "so I started finding out what I could do with WPS from Object REXX. I shouldn't be pumping application data into the user's desktop. Ideally, I should build a GUI application that opens its own window, and then populate that with the car dealer data. The problem is, Object REXX doesn't have a built-in GUI builder, and I'm not sure which of the GUI builders could handle tree-structured views of icons in a container. The WPS can do those things, so I used it."

"Yeah, it's a real pity about the lack of a built-in Object REXX GUI builder." Curt agreed.

"I used to think so too, but I'm not so sure anymore," said Steve. "I started reading a really great book last night. It's called *The Essential Distributed Objects Survival Guide*." [See reference in *Related Publications* on page xxvii.]

"Oh no, not another book!" Curt interjected.

"Relax, Curt," said Steve. "This one is by Bob Orfali and Dan Harkney—you know, the guys who wrote *Client/Server Programming with OS/2 2.1*."

"Oh yeah—well that one's OK; it's got lots of sample code," Curt admitted.

"So you don't mind books, so long as they have lots of pictures," Steve sniped.

"I'll say it's OK!" said Hanna. "That book has been our bible for coding OS/2 client/server applications over the last two years."

"Right," said Steve. "Well, in this new book about distributed objects, they paint a very different picture of how applications will be built in the future. They point out that with the availability of products like SOM, which allow separately built objects to talk to each other, it's possible to build complete applications just by collecting a whole lot of smart objects and wiring them together, in the same way that hardware designers can buy standard electronics components and wire

them together to build appliances. They call these smart objects *components*, and they predict that in the near future, most application developers will be building smart components that can be reused in many different applications.”

“What does ‘wiring objects together’ mean, Steve?” asked Hanna.

“Well with SOM, any object written in any language can send messages to any other object written in any other language,” Steve explained. “And SOM provides bridges between computers, so the objects can talk to each other, even if they’re running on machines with different operating systems in different locations. And it goes further than that. SOM is just IBM’s implementation of the CORBA standard. Lots of other vendors have built their own implementations of this standard, so SOM objects can talk to objects built using other vendors’ software. It’s the closest thing to universal middleware that there’s ever been.”

“Wonderful,” said Curt, “but what’s the connection with GUI builders?”

“Well—suppose we were in the business of developing world-class GUI builders for programmers,” said Steve. “Our biggest problem would be deciding which development language to support. There are GUI builders for C, C++, Smalltalk, Basic, Pascal, COBOL, PL/I, ...”

“OK, OK, there are lots of languages,” Curt interrupted, “so we’d have lots of choices. What’s the problem?”

“If we wanted to build the best GUI builder in the world and sell it to everyone, which language would we support?” Steve asked. “The market is hopelessly divided among all the languages that programmers use today. Pick any one language and you automatically exclude 80% of the potential market. Pick the top five languages and you might get 60% market coverage, but at the cost of supporting five versions of your GUI builder. And that’s really expensive.”

“Yeah,” said Curt, “but that’s just the way things are. There’s no way around it.”

“Until today,” Steve countered. “Suppose we built a GUI builder that is language-neutral. The programmers would use it to build all the panels and controls they need. But then, instead of writing code *inside* the GUI builder, they hit the **Generate** button, and the GUI builder turns their design into a SOM object that programs written in *any* language can drive, providing they have SOM support.”

“I get it,” said Hanna. “So the GUI object that you build isn’t tied to only one language. It would be just like the OS/2 WPS. You can drive it with scripts written in any language or even a mixture of different languages, if that’s what’s needed.”

“Exactly!” beamed Steve.

"That's pretty flexible," said Curt, "but it wouldn't sell. The real advantage of using a language-specific GUI builder like Dr. Dialog is that when a programmer builds a GUI control and he or she wants to link some code with it, all that has to be done is to double-click on the control and a REXX language editor window will open up. That makes it really easy to tie the code to the control that needs it. Your approach would take us back to the bad old days when all the code was dumped into one big bucket, and the programmer had to search through it to find which piece of logic related to which control."

"Not necessarily," was Steve's reply. "Suppose we had an application development framework..."

"What's a framework?" asked Curt.

"You should read the book—it's all in there," Steve answered. "But for simplicity, let's say a framework is a big component that has slots into which you can plug smaller components. An application development framework wouldn't have any built-in GUI builder or language editor or compiler or linker. It might just have a toolbar, and it would allow you to pick the components you want, like GUI builders, and drop them into its toolbar. When you want to build a GUI panel, the framework will launch the GUI builder you chose. And once you've edited a GUI control and you double-click on it, the GUI builder will send a message back to the framework, together with context information, to say what you clicked on. Then the framework will send the message on to the language editor that you chose, and it will open an edit window and show you the code that's currently linked to the control you clicked, if any, or else an empty panel. So far as the programmer is concerned, it all looks like one integrated application, but the integration takes place only on the glass. Behind the glass, you've got a bunch of separate objects exchanging SOM messages with each other."

A look of excitement had come to Hanna's face. "And the people who build those smart objects no longer have to do the whole development job," she said. "They can concentrate on building an object that does just one thing and does it really well."

"That's right," Steve agreed. "Programmers will be able to shop around for the components that best meet their particular needs, then plug and play the mix of components they've chosen. They'll all work together within a common framework."

"Where does Object REXX feature in this picture, if at all?" asked Curt.

"Anywhere!" said Steve. "Nobody will know or care what language the components are written in. Any language that can talk SOM is a first-class citizen in the new world of component architecture. One of the biggest strengths of REXX has always been its ability to act as glue. You can use it almost anywhere, anytime, to get different things working together. You don't need a compiler or an elaborate programmer's workbench. I think that Object REXX is going to turn in a star performance in pulling other components together to get the job done fast."

“So maybe we backed the right horse when we pinned our company’s future on Object REXX,” said Hanna. “Steve, this is awesome! Is all this in the book you spoke about? The one about distributed objects?”

“Part One of the book talks about this kind of approach in general terms,” said Steve. “It doesn’t deal with application development or any other application area in particular. The subsequent parts talk about the standards and products that will be used to make it happen. Things like SOM and OpenDoc and CORBA and OLE and COM.”

“So did you work out this wonderful approach to application building just from the principles they describe?” asked Hanna. “That’s pretty smart!”

“Oh, well, they did all the hard work and spelled out the basic ideas,” said Steve, trying unsuccessfully to look modestly heroic. “I just extended those ideas to cover the area of most interest to us—application development.”

“Hmph!” said Hanna, a small smile curving her lips. “Next you’ll be quoting Isaac Newton and saying, ‘If I have been able to see a little further than other men, it is only because I stood on the shoulders of giants’.”

“Not quite his style,” said Curt with a chuckle. “He’s more likely to say, ‘If I have been able to see a little further than other men, it is only because I stood on the toes of pygmies!’”

Curt and Hanna doubled up in laughter. Steve was torn between laughter and pique, but Hanna reached an arm around his shoulder and drew him close into the circle of mirth. All his resentment melted away, and he joined in the laughter, which continued for some time. Their company had faced many dangers in its short life, being often close to failure. Their decision to use first REXX and now Object REXX was really starting to pay off. They could feel the old tensions fading away, and a new sense of security replacing them. There also seemed to be the promise of exciting new things to come. At least for now, life was good.

12

Object REXX and the World Wide Web

The World Wide Web (Web) on the Internet is fast becoming the platform of choice for advertising applications. Therefore, in this chapter, let's rewrite the car dealer application to run on a Web server and use any Web browser as the GUI.

With minimal effort, we can port the car dealer application to run under the control of a Web server, using DB2 as the database. We can redesign the user interface, using the Hypertext Markup Language (HTML). The car dealer application creates most of the HTML documents from the data stored in DB2, using Common Gateway Interface (CGI) programs written in Object REXX.

Hacurs Connects to the Internet

It was after the long Labor Day weekend when Steve walked into the office with an unhappy expression on his face, seemingly carrying the weight of the world on his shoulders.

“What’s going on with you?” asked Hanna, concerned.

“Now that we’ve implemented the application for both Classy Cars and Value Vans, we are simply not busy enough” Steve replied. “We need to advertise our skills and our beautiful application, so that we get more companies interested in our services. I just have not figured out a good way of doing it.”

Curt, who had listened half-heartedly to the conversation, suddenly got up from his chair and shouted “The Internet!”

“The Internet?” asked Steve.

“Yes, the Internet,” reiterated Curt. “I visited the Computer Software Exposition at the Convention Center over the weekend, and lots of companies advertised their services and applications using one of those Web browsers connecting to their main home site.”

Hanna was silent for a moment, reflecting on what she had just heard. Then she said, “I think that’s a great idea, Curt. I have read so many articles lately about the Internet and the World Wide Web; we need to get our act together and become part of this exciting new technology.”

“What does it all take, Curt?” Steve asked, a little shyly. He felt badly that he did not really know much about the Web.

“Let’s sit down and make a list,” suggested Hanna. “Curt, you lead the discussion; of the three of us, you know most about the Web.”

Hacurs Makes a Plan for the Web

Curt got up from his chair, grabbed a marker pen, and marched to the flipchart stand. “Let’s see,” he began. “There are several things we have to do.

- [1] “First, we must physically connect our server to the Internet. That’s usually done through a high-speed leased phone line provided by the phone company. Then we need a modem at our end of line. We connect TCP/IP to the modem and line, using the Serial Line Internet Protocol (SLIP) or Point-to-Point Protocol (PPP).”

“Our line traffic will not be very big, I guess?” asked Hanna.

“True,” replied Curt. “We can get by with a medium-speed phone line for quite a while.”

“Luckily, we’ve already configured our LAN with TCP/IP,” said Steve. “It should be a breeze to connect our desktop machines and the ThinkPads to the Internet through our LAN server.”

“There you go, Steve,” laughed Curt.

- [2] “Second, we have to install an Internet server program on our LAN server. Many server products are on the market, but for our OS/2 system I think the best server is the IBM Internet Connection Server for OS/2. I saw a demonstration at the exposition in the IBM booth. IBM currently has a promotion, and it takes only a few minutes to download the server for free from an IBM site.”

“Is that server hard to install?” asked Steve.

“It looks very easy,” replied Curt. “There is only one configuration file to be updated with our installation-specific information, and the product even provides a Web browser dialog to do most of the tailoring.”

“I guess we have to install one of those Web browsers” added Hanna.

- [3] “You’re right, Hanna” responded Curt. “That is the third point: a Web browser on every machine. And for our OS/2 machines, the best browser is the IBM WebExplorer.”

“I have read so much about that other browser, but I don’t recall its name,” said Hanna. “Aren’t the other browsers better?”

“Not for OS/2!” Curt shouted a little angrily. “The other browsers run mostly under Windows. They can be run in an WINOS2 session, but then they run in 16-bit mode, whereas the WebExplorer is a 32-bit application for OS/2.”

“Great,” intervened Steve. “We’ll stick with OS/2; it’s been a wonderful product for our needs. What else do we need, Curt?”

“What does a user see when he connects to our server?” asked Hanna.

- [4] “That’s item number four: a home page,” Curt said. “The home page is the first thing you see when you point your browser to a Web server. Our home page must make a statement about our company that entices people to want more information about us. It has to be attractive and lure users into our net—the car dealer application.”

“I’ll help you design the home page,” said Hanna. “We can use our logo, add some information about ourselves, and then go directly into advertising the car dealer application.”

“That sounds wonderful, Hanna,” Curt agreed.

“By the way, how is a home page designed?” asked Steve.

Curt explained: “All Web pages are written as a file with the Hypertext Markup Language, or HTML for short. It’s a tag language, similar to some word processors. There are many tools on

the market to design Web pages interactively in WYSIWYG mode, then generate the HTML file. Our home page is probably simple enough to just code directly in HTML.”

“Don’t we have to create the Web pages for the car dealer application from the data stored in DB2?” asked Steve. He was now very interested in understanding and learning more about Web technology.

- [5] “Yes,” said Curt. “And this leads directly to the fifth item—the car dealer application. We have to design the flow of how a user can look at the car dealer data. Then we design each of the pages individually and write an Object REXX program to generate the page.”

“How are these programs invoked?” asked Steve.

“Most Web servers support the Common Gateway Interface, or CGI for short,” replied Curt. “In the configuration file, you specify which requests should be handled by a program, as opposed to just returning a predefined HTML file. The program can create the HTML file on disk and tell the server about it or, for better performance, it can pass the lines of the generated HTML page directly to the server. Most servers pick up the output by rerouting standard output, so we just use the Object REXX *say* instruction to prepare the pages.”

“That sounds easy enough for me,” Steve added. “I’ll work on that because I understand the DB2 database the most. The hard part will be to learn the syntax of HTML. I had better go to the bookstore to buy a manual.”

“I bet you will find a great way to generate those pages from DB2,” Hanna joked. “Maybe, you’ll even define an Object REXX class to handle the HTML easily!”

“Hmm, not a bad idea from a young kid like you!” Steve replied, and he started to leave.

- [6] “Don’t run away yet!” said Curt, holding Steve back by the arm. “We have to decide on an Internet name for our server and register it with the gods of the Internet.”

“What does an Internet name look like?” Hanna asked.

“Well, it’s something like ‘www.ibm.com,’ so I suggest we name our Internet server ‘www.hacurs.com,’ and our machines could then have the names ‘steve@hacurs.com’ and so forth.”

“I like it!” Steve exclaimed, and Hanna agreed, as well, after deeply pondering her new name.

“OK Curt, you and I can install the Internet server and WebExplorer this afternoon,” she said. “We can work with them on our existing LAN without the leased line for now. Then we’ll meet tomorrow morn-

ing to work on the home page. Steve, you can go to the bookstore and get us some manuals about HTML and the CGI way of invoking programs. I think we're on a roll!"

Hanna was happy to see that Steve was excited about the World Wide Web. He had been morose for quite some time, but now his face was lit up, and he was ready to tackle any problems that might arise. Indeed, everything looked bright again.

Hacurs Designs a Home Page

The next morning, Hanna and Curt worked together to design the home page. Both were eager to get it done quickly.

"We don't have to design the *ultimate* home page," Hanna reasoned. "It should be simple, not too long, and provide the essential information about our company. What information do we need on it?"

"Let's make a list," suggested Curt.

They brainstormed for 15 minutes and came up with the following topics.

- ☐ Company logo
- ☐ People (Hanna, Curt, Steve), with some personal details
- ☐ Services offered
- ☐ Introduction to the car dealer application

"The company logo we can enter as is, because Web browsers handle all kind of graphic files," Curt explained. "For the people, we use a table with our names and personal details. Then we list our services in boldface, and for the car dealer we could use some of the cute little icons we developed for the WPS application. One of these icons will start the application."

"We could make the table a little more interesting by adding a picture of our car to each row. I mean, we do have that neat camera to take electronic pictures!" Hanna suggested.

"That's a good start; let's go to work," said Curt, who was getting eager as well, and the HTML manuals Steve bought from the bookstore were all ready to get dirty.

The Home Page

Within a few hours, Hanna and Curt managed to get the home page coded in HTML. The table of the people was a little tricky, but after some trials the home page [Figures 62 and 63] saw the light of the day on Curt's ThinkPad.

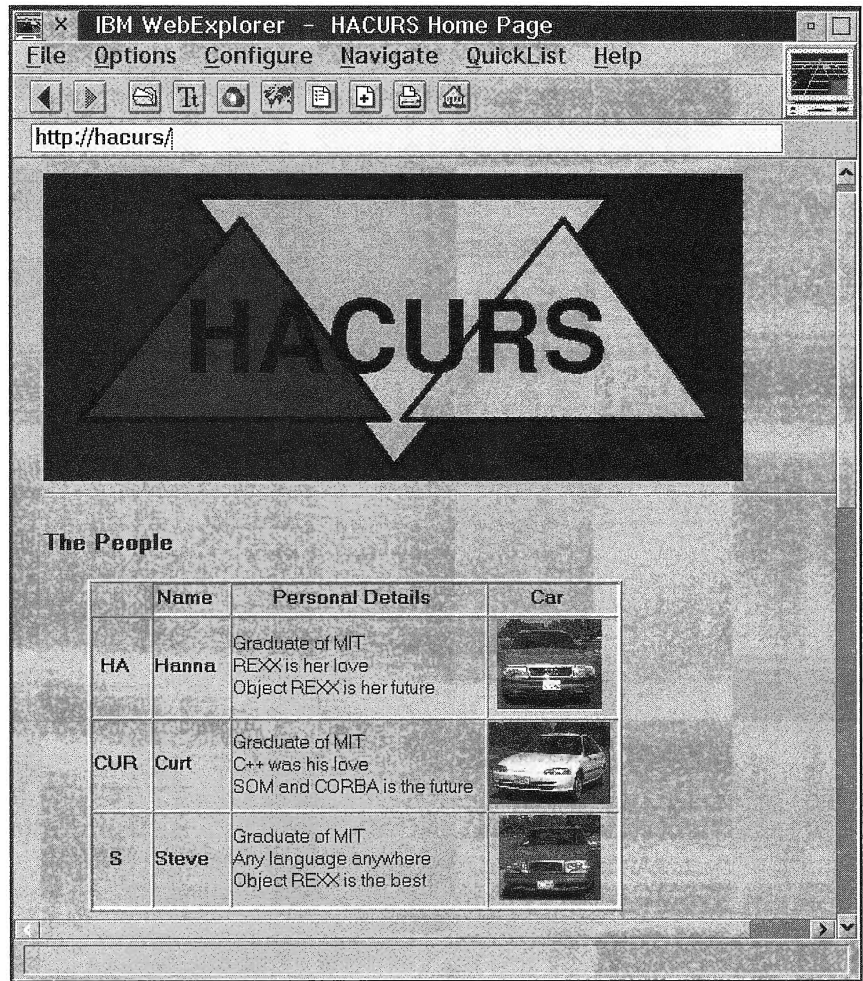


Figure 62. *Hacurs Home Page: Top Half*

Notes:

To connect to the home page of the Hacurs company with a Web browser, you would enter:

`http://www.hacurs.com/`

or

`http://www.hacurs.com/Hacurs.htm`

We used the TCP/IP *HOSTS* file to make a shorthand entry, *hacurs*, to point to the OS/2 machine with the Internet Connection Server.

`129.33.160.207 www.hacurs.com hacurs`

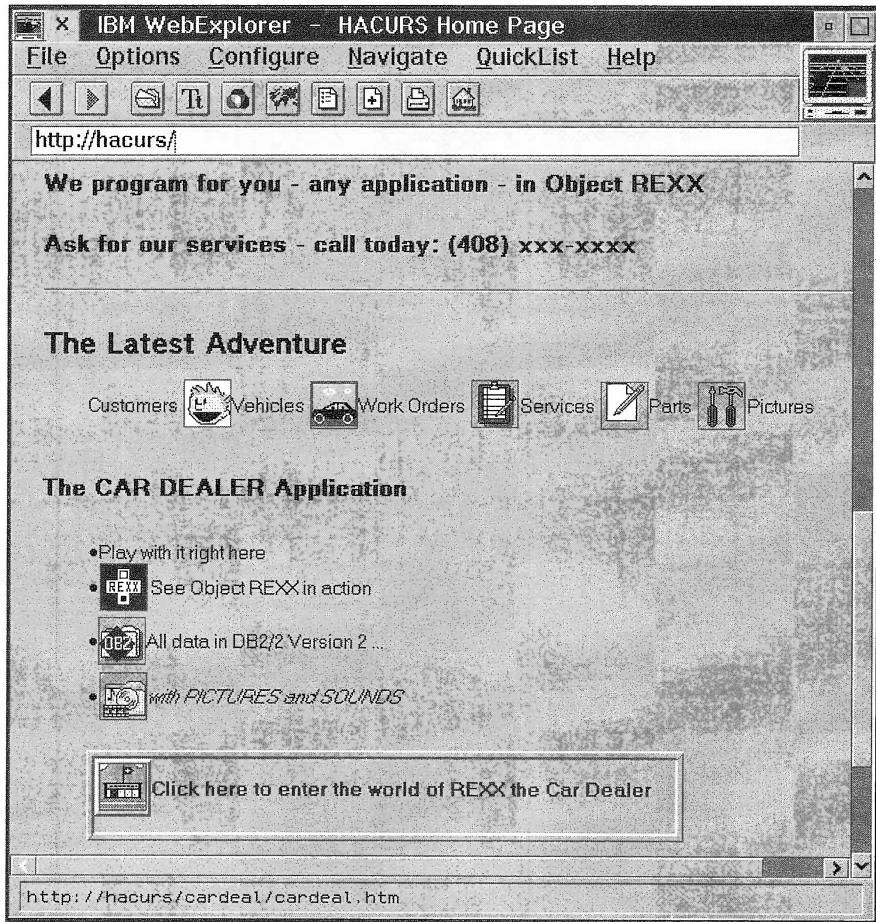


Figure 63. *Hacurs Home Page: Bottom Half*

Those readers who want to know what the Hacurs' home page looks like in HTML can see the actual coding in Figure 64. The home page is usually stored in the HTML directory of the Web server.

```
<!------->
<! WWW\Hacurs.htm      CarDealer - Web - Hacurs Home Page  ITS0-SJC ->
<!------->
<html> <head> <title> HACURS Home Page </title> </head>
<body>

<hr>
<h3> The People </h3>
<dir>
```

Figure 64. *(Part 1 of 2) Hacurs Home Page HTML Code*

```

<table border=2 cellpadding=0>
<tr>
<th> </th> <th> Name </th> <th> Personal Details </th> <th> Car </th>
<tr>
<td align=center> <b> HA </b> </td> <td align=left > <b> Hanna </b> </td>
<td align=left > <br> Graduate of MIT      <br> REXX is her love
                        <br> Object REXX is her future                                </td>
<td align=center>  </td>
<tr>
<td align=center> <b> CUR </b> </td> <td align=left > <b> Curt </b> </td>
<td align=left > <br> Graduate of MIT      <br> C++ was his love
                        <br> SOM and CORBA is the future                                </td>
<td align=center>  </td>
<tr>
<td align=center> <b> S </b> </td> <td align=left > <b> Steve </b> </td>
<td align=left > <br> Graduate of MIT      <br> Any language anywhere
                        <br> Object REXX is the best                                </td>
<td align=center>  </td>
<tr>
</table>
</dir>
<p>
<hr> <h3> We program for you - any application - in Object REXX </h3>
      <h3> Ask for our services - call today:  (408) xxx-xxxx </h3>
<hr>
<h1> The Latest Adventure </h1>
<dir>  <p> Customers    
        Vehicles    
        Work Orders 
        Services    
        Parts       
        Pictures
      <p>
</dir>
<h3> The CAR DEALER Application </h3>
<ul>
  <li> Play with it right here
  <li>  See Object REXX in action
  <li>  All data in DB2/2 Version 2 ...
  <li>  <em> with PICTURES and SOUNDS </em>
</ul>
</ul>
<dir>
  <table border=4 cellpadding=0> <br>
    <a href="/cardeal/cardeal.htm">
      
      <strong> Click here to enter the world of REXX the Car Dealer </strong> <
  </a>
  </table>
</dir>
<hr> <b> HACURS </b>
      <address> swiss@hacurs.com  <br> (408) xxx-xxxx </address>
      <p>
      <b> Ulrich (Ueli) Wahli - IBM ITSO San Jose </b>
      <address> wahli@vnet.im.com </address>
<hr>
</body></html>

```

Figure 64. (Part 2 of 2) Hacurs Home Page HTML Code

Note: Web browsers compress multiple blanks to single blanks, and lines are concatenated unless a tag forces a new line.

Web Car Dealer Application

In the meantime, Steve had designed the car dealer application for the Internet. On a few sheets of paper, he sketched out the different formats for presenting the car dealer data in Web browser pages. He called Curt and Hanna over to his desk and showed them his initial design [see Figure 65].

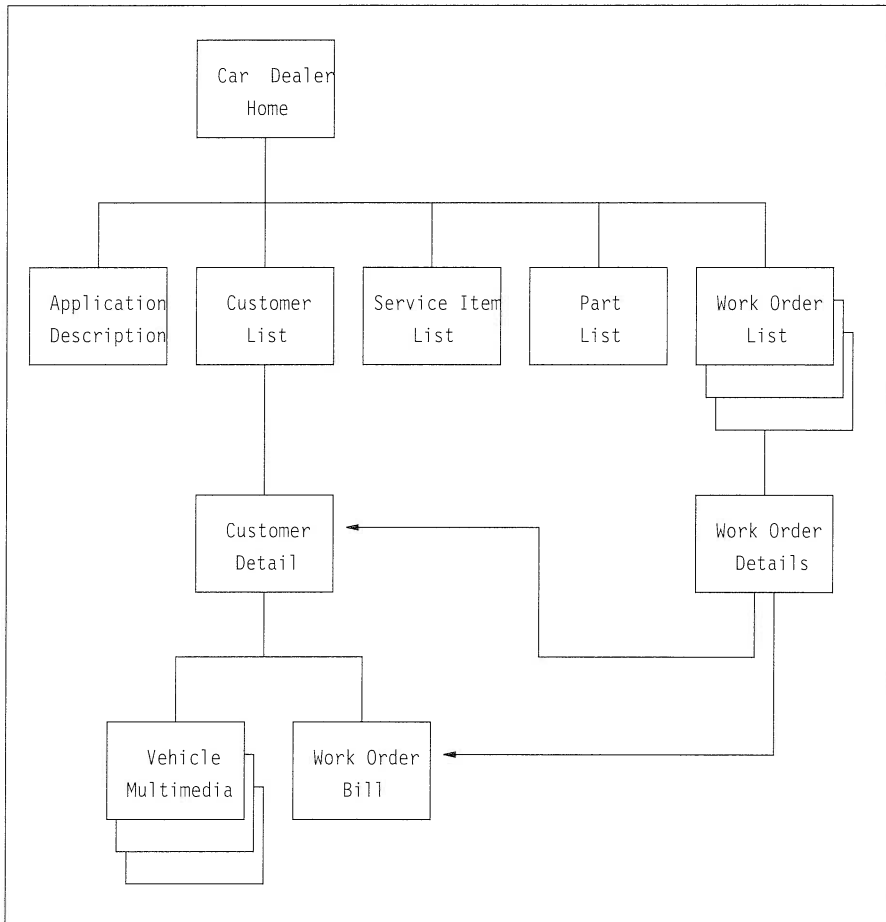


Figure 65. *Initial Design for the Car Dealer Application on the Web*

Steve explained: “I start with a home page, from where we can invoke the different paths. The path most used will be the customer list. We provide a customer search facility by partial name, as we did in the GUI applications.” [See Figure 13 on page 83.]

“Customer search presents a list of matching customers, from where we can invoke the details of one customer. The details will include all the cars of the customer, the work orders for each car, including the list of services and parts of the work order,” Steve continued.

“Then you can invoke the bill for a selected work order,” said Hanna.

“That’s right,” Steve responded. “And for our multimedia *New and used cars* customer, we can display the pictures or play the sounds and videos. I represented that with the three stacked boxes. The other paths are to list all service items, parts, or work orders, and I think we should also have a short application description.”

“Why do you show three stacked boxes for *Work Order List*?” asked Curt.

“Ah, yes, I forgot to mention that,” said Steve. “I designed it so that we can list incomplete work orders, complete work orders, or all work orders. Remember, we have that search facility implemented in the *Work Order* class. When a work order is selected, I will show the details, including the service items with parts, and the customer and vehicle. From there, the user can get the customer details or the bill.”

“That all looks very good,” said Curt. “When can we see it running?” he asked, smirking.

“I will start coding a simple page first, such as the *Part List*. Once I get familiar with the CGI technique of invoking an Object REXX program, it should be a breeze to get the other pages done. Remember, the object model is working and stable. Retrieving the data from DB2 is simple; we already have all the methods. It is just a matter of accepting the parameters from the Web browser and creating the HTML output.”

“Show us your first page tomorrow—I have lots of confidence in you,” said Hanna, smiling and she left. Steve just stood there, but what Hanna had said filled him with pride. He would have something running by the morning.

Web Common Gateway Interface

Steve studied the documentation of the Internet Connection Server carefully. He found that the way to invoke a CGI program was by entering the Web browser request as:

```
http://hacurs/cgi-bin/progname?parms
```

This would invoke the program *progname* in the CGI-BIN subdirectory of the server (d:\WWW\CGI-BIN). The program could be either a .EXE or .CMD file. The parameters would not be passed directly to the program. They would be stored as OS/2 environment variables, together with other useful information about the request [Figure 66].

REMOTE_ADDR	TCP/IP address of the requester (xxx.xxx.xxx.xxx)
SCRIPT_NAME	Request string before the ? (/cgi-bin/progname)
QUERY_STRING	Parameters following the ? in the request (parms) (blanks are replaced by + signs)
...	

Figure 66. CGI Environment Variables (Extract)

Steve proceeded to write the first program to list all parts in the database. All he had to do was to initialize the application, output the top part of the HTML file, iterate through all the parts and output each part in HTML, conclude the HTML file, and close the application. He decided to use an HTML table to display the part list [Figure 67].

```

/*-----*/
/* WWW\partall1.cmd      CarDealer - Web - Part list 1      ITS0-SJC */
/*-----*/
.Cardeal~initialize
partclass = .local['Cardeal.Part.class']
say 'Content-Type: text/html'
say ''
say '<html>'
say '<head><title>Object REXX Car Dealer Application</title></head>'
say '<body>'
say '<H2>Part List</H2>'
say '<table border=2 cellpadding=0>'
say '<tr>'
say '<th>Number</th> <th>Description</th> <th>Price</th> <th>Stock</th>'
say '<tr>'
do part over partclass~extent
  say '<td>' part~number '</td>'
  say '<td>' part~description '</td>'
  say '<td align=right>' part~price '</td>'
  say '<td align=right>' part~stock '</td>'
  say '<tr>'
end
say '</table>'
say '</body>'
say '</html>'
.Cardeal~terminate
return
::requires carmodel.cfg      /* include the configuration file

```

Figure 67. CGI Program to List All Parts

Steve ran the program and was pleased with the output. He called Hanna and Curt and showed them how simple the program was.

“This looks really easy!” Curt said in astonishment. “But what are those first two *say* instructions?” he asked.

Steve explained: “A CGI program must first tell the server what kind of output is produced. The string *Content-Type: text/html* tells the server that a regular HTML file will be generated, and the second *say* instruction must be blank.”

"I thought you were going to write an HTML class to simplify the coding of the generated HTML lines," Hanna interjected.

"That's true," said Steve. "But first I wanted to have a simple working example. Now I can design the HTML class to provide the functions that are used most."

"Show us the output in the browser," demanded Curt, who was excited and wanted to see the program in action.

"Here we go," said Steve as he started the WebExplorer and entered `http://hacurs/cgi-bin/partall1`. It took a while, but eventually the screen filled with the Part List [Figure 68].

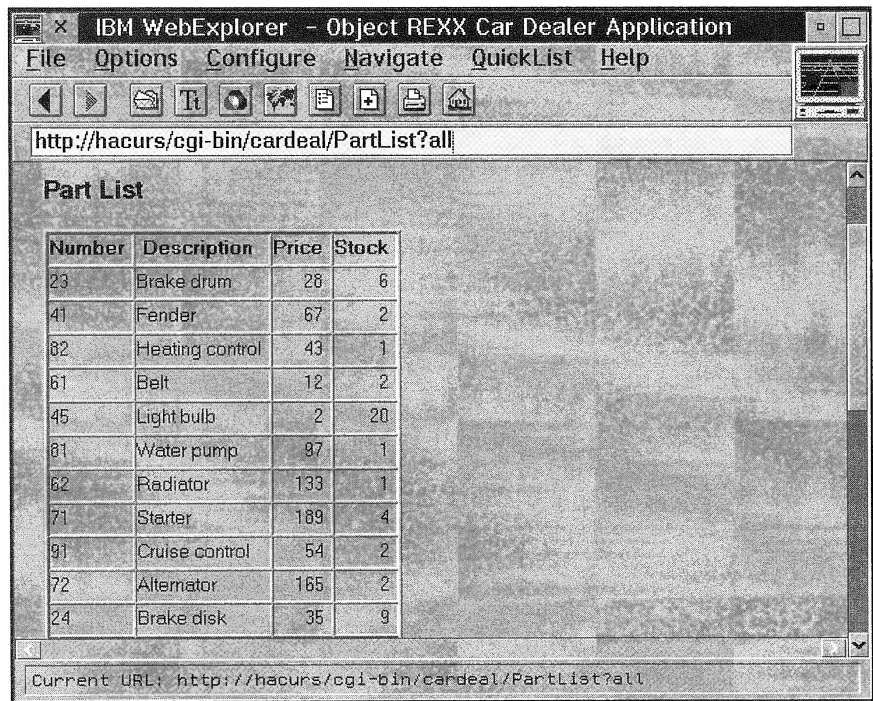


Figure 68. Car Dealer Part List in Web Browser

HTML Class

After a nice lunch at the nearby Mexican cantina, Steve proceeded to rewrite the code using a new HTML class. He thought of the functions that are used most often and designed those as methods of the class.

He also decided that each car dealer output page should have a reference to the car dealer home page and a common signature area at the bottom. The redesigned code looked definitively more object-oriented [Figure 69].

```

/*-----*/
/* WWW\partall2.cmd      CarDealer - Web - Part list 2      ITS0-SJC */
/*-----*/

.Cardeal~initialize
partclass = .local['Cardeal.Part.class']
html = .HTML~new
html~title('Object REXX Car Dealer Application')
html~carhome          /* reference to car dealer home page */
html~h2('Part List')
html~table('border=2 cellpadding=0')
html~tr
html~th('Number')~th('Description')~th('Price')~th('Stock')
html~tr
do part over partclass~extent
    html~td(part~number)~td(part~description)
    html~td(part~price,'align=right')~td(part~stock,'align=right')
html~tr
end
html~etable
html~p~carhome          /* reference to car dealer home page */
html~sign              /* common signature at bottom      */
html~send              /* output all the accumulated lines */
.Cardeal~terminate
return
::requires html.frm
::requires carmodel.cfg

```

Figure 69. *Object-Oriented CGI Program to List All Parts*

The `HTML` class allocates an array of lines. Each method basically adds a line to the array in the proper HTML format. Some of the methods produce matching start-and-end tags, with the argument passed as the text between the tags. For example:

```
html~h2('Part List') ==> <H2>Part List</H2>
```

Other HTML tags are produced by individual start-and-end methods:

```
html~table('border=2 cellpadding=0') ==> <table border=2 cellpadding=0>
html~etable ==> </etable>
```

The `title` method produces all of the required HTML tags at the start of the document:

```
html~title('xxxxxx') ==> <html> <head> <title>xxxxxx </head> <body>
```

The `carhome` method produces the reference to the car dealer home page, the `sign` method produces the common ending, and the `send` method outputs the whole array as REXX *say* instructions.

Not every HTML tag has a matching method. Tags without a method can be generated with generic methods, where the name of the tag is passed, as well. [See Figure 70 for an extract of the `HTML` class.]

```

/* WWW\html.frm          CarDealer - Web - HTML Framework      ITS0-SJC */

::class HTML public subclass array

::method init                                /* initialize an html object      */
  expose array_index type                    /* index into the array, docu type */
  array_index = 1                           /* start at the first item         */
  type = 'text/html'                        /* default document type          */
  forward class (super)                     /* do superclass initialization   */
  /* Start the html array off              */
::method put                                /* over ride of the put method    */
  expose array_index                        /* get the current index          */
  parse arg text
  self~put:super(text, array_index)
  array_index = array_index + 1
::method title                              /* title tag                      */
  parse arg text
  self~put('<html><head><title>'text'</title></head><body>')
::method h1                                /* header 1 tag                  */
  parse arg text
  self~put('<H1>'text'</H1>')
::method tag 1                             /* generate any tag              */
  parse arg name, text
  self~put('<'name'>'text')
::method text                              /* add raw text to the stream    */
  parse arg text
  self~put(text)
::method p                                /* paragraph tag                 */
  parse arg text
  self~put('<p>'text')
::method ul                                /* ul tag                        */
  self~put('<ul>')
::method li                                /* li tag                        */
  parse arg text
  self~put('<li>'text')
::method table                             /* table tag                     */
  parse arg options
  self~put('<table' options'>')
::method td                                /* td tag                        */
  parse arg text, options
  if text = '' then self~put('<td' options'>')
  else self~put('<td' options'>'text'</td>')
::method sign                              /* signature/end                 */
  self~hr~b('Hacurs - Car Dealer Application')
  self~br('Ulrich (Ueli) Wahli - IBM ITS0 San Jose')
  self~address('wahli@vnet.im.com')
  self~hr~etab('body')~etab('html')
::method send                              /* send the HTML from the array  */
  expose type
  crlf = '0d0a'x
  say 'Content-Type:' type
  say ''
  say '<!doctype html public "html2.0">'
  do line over self                        /* loop over the array          */
  say line                                /* send out the next line       */
end

```

Figure 70. HTML Class for CGI Programs (Extract)

Customer Search Form

The next morning, Steve showed the HTML class to Hanna and Curt. “This will make future coding much easier,” he explained.

“That’s true,” said Hanna. “But how are you going to implement the customer search facility? Can you put a push button into an HTML page?”

“I already investigated that last night,” Steve said. “HTML provides the form facility with entry fields, radio buttons, and check boxes, and a **Submit** button to pass the values of the form to the next CGI program. The extract of the customer home page for customer search looks like this.” [Figure 71.]

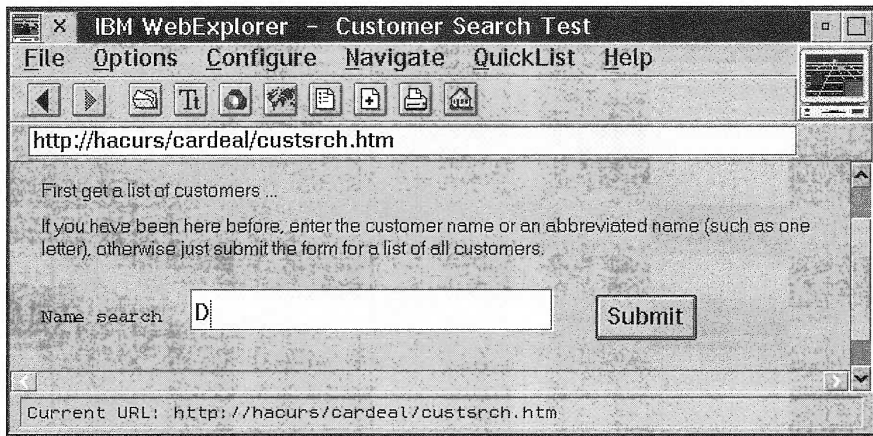


Figure 71. *Customer Search Form*

“Explain this one to me, please,” said Curt.

“Sure, I can do that,” Steve answered. “Just look at the HTML code that creates this form.” [Figure 72.]

```
<html> <head> <title> Customer Search Test </title> </head> <body>
<form method="GET" action="/cgi-bin/CustList">
  <p> First get a list of customers ...
  <p> If you have been here before, enter the customer name or
    an abbreviated name (such as one letter),
    otherwise just submit the form for a list of all customers.
  <p> <pre>Name search   <input name="name" type="text" size="20">   <input type="submit">
    </pre>
</form>
</body> </html>
```

Figure 72. *HTML for Customer Search Form*

“The *form* tag defines the method of passing data and the program, *CustList*, that is invoked. The *GET* method passes all data in the request string, whereas the *POST* method tells the program to retrieve the data from the browser once it has been invoked. I specified the *GET* method because the amount of data is small.

“The input tags specify the different fields and buttons of the form. Here I used only one input field *name* of 20 bytes, and one **Submit** button. The `<pre>` tag specifies that this line is preformatted, with blanks between the text label, the input field, and the **Submit** button. Normally, Web browsers reduce all blanks to a single blank,” Steve concluded.

“What does the program get passed when I click on the **Submit** button?” asked Hanna.

“The browser builds a query string from all of the fields of the form. Each field is passed in the format *fieldname=value*, separated by an ampersand. In our simple form with one field,

`http://hacurs/cgi-bin/CustList?name=D`

“would be the request string if you enter *D* in the search field. If the form had a name and an address field, the request string would include both fields, separated by an ampersand.”

“Come see me after lunch, and I’ll show you the customer search in action,” said Steve, who was confident that it would be fairly easy to write the second program, using the form and his new HTML class.

After lunch, Steve showed the customer list output to Hanna and Curt. He had designed a table to hold the customer data. He made the customer names active so that clicking on a name would invoke the next program, *CustDetail*, to generate the customer detail page. [See Figure 73.]

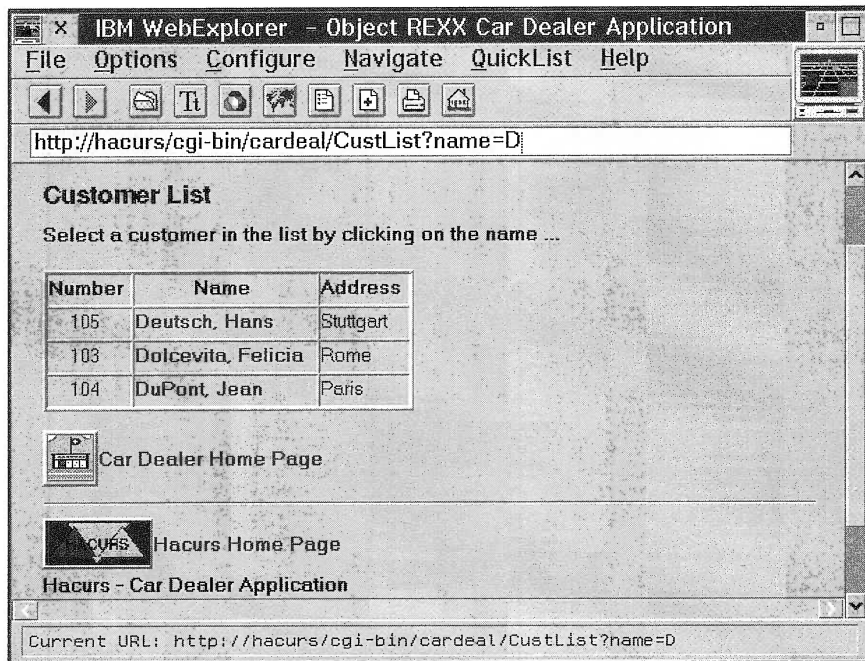


Figure 73. *Customer List in Web Browser*

"Wow, you really did a lot of work!" exclaimed Hanna, who was very pleased with the progress Steve had made. The other pages would be fairly easy to add. The work on the object model and the configuration paid off with every new application based on that model.

"Have you tried to run the application with persistent storage in files?" asked Curt.

"No problem," replied Steve. "I am doing most of the test with the file system because it is faster than the DB2-based application. I just switch the configuration file [carmodel.cfg], using our car-run program. But when we make the Web application available to outside users, it is better for advertising if it runs on DB2."

Steve had another ace up his sleeve. He clicked on a customer name in the list, and the details of the customer showed up in the WebExplorer. [See Figure 74.]

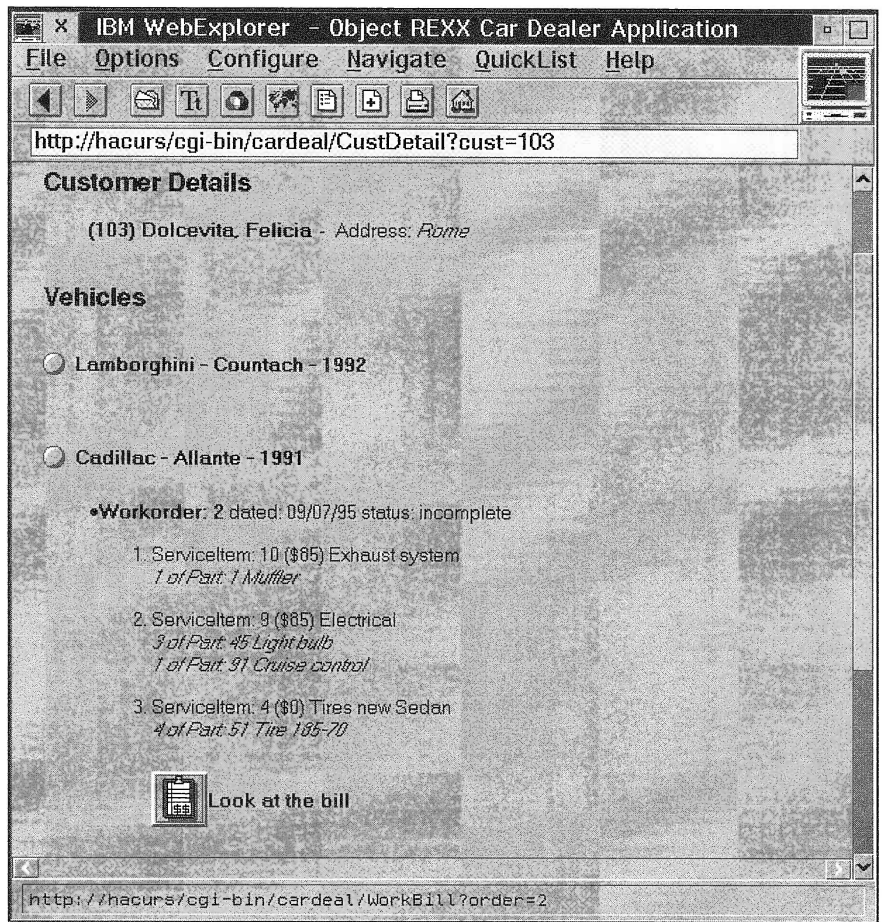


Figure 74. *Customer Details in Web Browser*

“What else is there to do?” asked Curt. “Just a few more programs generating the other Web pages.”

“I think there are a few more items on my list,” Steve replied.

Program Organization and Performance

“The car dealer is just the first application we put on the Internet. In the future, we might add other applications. I have to organize my files better, so that future applications do not interfere with the car dealer,” Steve continued.

“Steve, that’s good thinking ahead,” said Hanna. “The Internet could be useful for many things we do over the next few months. We had better start organizing all the programs and HTML files we produce for the car dealer application.”

“There is another thing that bugs me,” Steve added. “In every CGI program, I have to initialize the application and terminate it afterward. That adds a lot of overhead and makes the application look slow. I must find a way of keeping the object classes in memory so that each CGI program can immediately access them. I have an idea, but I have to run a few tests to make sure that the design holds.”

Customizing the File Organization on the Web Server

Steve decided to put all the car dealer HTML files and programs into a separate subdirectory. At first, he considered using a subdirectory within the Internet Connection Server directory structure but, after studying the documentation on server administration, he decided to use a subdirectory within the existing car dealer directory:

```
d:\CARDEAL\WWW
```

He then tailored the server administration file to point to the new directory. The `httpd.cnf` administration file is stored in the ETC directory of TCP/IP.

Note: Issue the *SET ETC* command to find the ETC directory. It is usually called either `d:\MPTN\ETC` or `d:\TCP\IP\ETC`.

Figure 75 on page 202 shows an extract of the tailored administration file.

“Why are you making these changes, Steve?” asked Hanna, glancing over Steve’s shoulder. She had just returned with coffee from the machine and wondered why Steve was so engrossed in his work.

It was as if Steve had just awakened. He had not realized that Hanna was standing right behind him. He started to apologize for not noticing her, but then he just shrugged and explained:

- ❶ “The first *Welcome* line directs the server to display the Hacurs home page:

```
http://www.hacurs.com                ==> d:\WWW\HTML\Hacurs.htm
http://www.hacurs.com/Hacurs.htm     ==> same
```

- ❷ “The second *Welcome* line directs the server to display the car dealer home page if the car dealer directory is selected:

```
http://www.hacurs.com/cardeal        ==> d:\CARDEAL\WWW\cardeal.htm
http://www.hacurs.com/cardeal/cardeal.htm ==> same
```

[**Note:** Point ❸ directs any requests beginning with `/cardeal` to the `d:\CARDEAL\WWW` subdirectory.]

```

#      Sample configuration file for Web Server for OS/2
#
.....
# added for car dealer application (next 2 lines)
1 Welcome      Hacurs.htm
2 Welcome      cardeal.htm
  Welcome      Welcome.html
  Welcome      welcome.html
  Welcome      index.html
  Welcome      Frntpage.html
.....
# added for car dealer application (next 4 lines)
3 Exec    /cgi-bin/cardeal/*      D:\CARDEAL\WWW\CGIREXX.CMD
4 Pass    /cardeal/media/*        D:\CARDEAL\Media\*
5 Pass    /cardeal/*              D:\CARDEAL\WWW\*
6 Pass    /tmp/*                  E:\TMP\*

Exec    /admin-bin/*  E:\WWW\ADMIN\*
Exec    /cgi-bin/*    E:\WWW\CGI-BIN\*
Pass    /Docs/*       E:\WWW\DOCS\*
Pass    /httpd-internal-icons/* E:\WWW\ICONS\*
Pass    /icons/*      E:\WWW\ICONS\*
Pass    /Admin/*      E:\WWW\ADMIN\*
Pass    /*            E:\WWW\HTML\*
.....

```

Figure 75. *Tailored Web Server Administration File*
Extract of the HTTPD.CNF file in the ETC directory.

- 3** “The *Exec* line invokes the CGIREXX.CMD program for every CGI request starting with */cgi-bin/cardeal*:

/cgi-bin/cardeal/progname?parms

“I plan to write one interface program that handles the environment variables and some housekeeping before invoking the individual function programs.

“I guess that putting common code into one CGI program will make the individual programs a little simpler,” remarked Hanna. Steve nodded and continued:

- 4** “The first *Pass* line directs the server to the Media subdirectory for any */cardeal/media* requests. We will need that to display the car pictures if we run with file persistence.

- 5** “The second *Pass* line directs any car dealer request to the WWW subdirectory.

- 6** “The last *Pass* line directs any requests for */tmp* to the temporary directory [as set in *SET TMP* in the CONFIG.SYS file]. That’s where the pictures are extracted to when we run with DB2.”

“I am impressed!” gasped Hanna. “You have thought of everything. This keeps all the car dealer files nicely separated from the normal Web server files.

Have you thought about performance yet?” she added. “How can you keep all of the class objects in memory?”

Optimizing Performance

“Here’s my idea,” Steve answered. “For every client program request, the server starts a thread. That’s where I initialize the car dealer application, generate the HTML file, and terminate the application. What I would like to do is initialize the application outside of the Web server, then just use the classes in memory from the Web programs.”

“I see your problem,” said Hanna. “Our application stores information in the local environment [see *Communication Among Classes* on page 152], and every class stores itself, too. These local variables are not available in another process.”

“What I have to do is write a small program that starts the application and stores the necessary information in the global environment [see *The Global Directory* on page 154]. Then I have direct access to all of the class information from my CGI programs,” Steve pondered.

“Good idea,” said Hanna. “And since you have already decided to have one main CGI program, *CGIREXX*, you can pick up the global information in that program and convert it to the local variables needed by the car dealer application.”

“I can also check whether the application is running and produce a nice error message if it is not,” added Steve.

“There you go, but it’s not an error message, it’s another Web page saying, ‘Sorry, the application is not running at this time.’ That is much nicer than an unfriendly error message,” said Hanna. “I’ll prepare that page for you, busy guy. I’ll name it *cardealN.htm*.”

Car Dealer Start Program for the Web

Steve quickly wrote the program to start the car dealer application [see Figure 76 on page 204]. He would enhance it later to provide more function and make it usable from a Web browser, as well.

```

/*      carstart.cmd      CarDealer - Web - Start Application      */
parse upper source env . me .
maindir = me~left(me~lastpos('\WWW\')-1)
curdir = directory()
x = directory(maindir)
call carmodel.cfg          /* configuration file */
.Cardeal~initialize
.environment['Cardeal.Data.type']      = .local['Cardeal.Data.type']
.environment['Cardeal.Data.dir']        = .local['Cardeal.Data.dir']
.environment['Cardeal.Media.dir']       = .local['Cardeal.Media.dir']
.environment['Cardeal.Customer.class']  = .local['Cardeal.Customer.class']
.environment['Cardeal.Vehicle.class']   = .local['Cardeal.Vehicle.class']
.environment['Cardeal.WorkOrder.class'] = .local['Cardeal.WorkOrder.class']
.environment['Cardeal.ServiceItem.class'] = .local['Cardeal.ServiceItem.class']
.environment['Cardeal.Part.class']       = .local['Cardeal.Part.class']
.environment['Cardeal.WorkServRel']     = .local['Cardeal.WorkServRel']
say
say 'Waiting for you to....'
say 'Press enter to stop Car Dealer Application'
pull ans
.Cardeal~terminate
.environment['Cardeal.Data.type'] = .nil
/* other variables similar */
x = directory(curdir)
return

```

Figure 76. *Car Dealer Start Program for the Web*

This is an abbreviated version of the program. The real program (carstart.cmd) uses a class with *start*, *display*, and *stop* methods.

Car Dealer Common Interface Program

Next, Steve attacked the common interface program, *CGIREXX*. He had to implement a number of common functions:

- ❑ Pick up the environment variables holding the request and the parameters from the Web server.
- ❑ Check whether the car dealer application is running. Return the Web page Hanna designed if the application was not available; otherwise, prepare the *local* variables.
- ❑ Invoke the individual program to handle the request. He decided to pass the same Web server environment variables to all programs, even if they were not needed.

The task was not too difficult, and soon Steve tested the new interface program shown in Figure 77.

```

/* WWW\cgirexx.cmd      CarDealer - Web - CGI REXX Interface      */
parse source env . me .
envir = 'OS2ENVIRONMENT'
sourcdir = me~left(me~lastpos('\')-1)
script = value('SCRIPT_NAME',envir)      /* Web server variables */
who      = value('REMOTE_ADDR',envir)
list     = value('QUERY_STRING',envir)
parse var script '/cgi-bin/' type         /* extract request type */
list=translate(list, ' ', '+')||'090a0d'x) /* Whitespace, etc.    */
ddir     = sourcdir                       /* CARDEAL\WWW directory */
x = directory(ddir)
sqlca.sqlcode = 0                        /* init DB2 return code */
.local['Cardeal.Data.type']              = .environment['Cardeal.Data.type']
.local['Cardeal.Data.dir']                = .environment['Cardeal.Data.dir']
.local['Cardeal.Media.dir']               = .environment['Cardeal.Media.dir']
.local['Cardeal.Customer.class']          = .environment['Cardeal.Customer.class']
.local['Cardeal.Vehicle.class']           = .environment['Cardeal.Vehicle.class']
.local['Cardeal.WorkOrder.class']         = .environment['Cardeal.WorkOrder.class']
.local['Cardeal.ServiceItem.class']       = .environment['Cardeal.ServiceItem.class']
.local['Cardeal.Part.class']              = .environment['Cardeal.Part.class']
.local['Cardeal.WorkServRel']             = .environment['Cardeal.WorkServRel']
if .local['Cardeal.Data.type'] = 'DB2' then do
  call sqlexec "CONNECT RESET"            /* just to be sure      */
  call sqlexec "CONNECT TO DEALERDB"      /* connect to database  */
end
select
  when .environment['Cardeal.Data.type'] = .nil then
    call returnfile ddir'\cardealN.htm' /* CAR DEALER NOT RUNNING */
  when sqlca.sqlcode \= 0 then
    call returnfile ddir'\cardealN.htm' /* DB2 DB CONNECT FAILED */
  when type='cardeal/cardeal' then
    call returnfile ddir'\cardeal.htm' /* cardeal home page    */
  when type='cardeal/CustList' then
    call custlist file, type, list, who
  when type='cardeal/CustDetail' then
    call custdeta file, type, list, who

  /* others similar ..... */

  otherwise
    call error /* not shown, returns an HTML error page */
end
return
/*----- return a precoded HTML file -----*/
RETURNFILE:
  parse arg resultfile
  say 'Location:' ,
    '/cardeal'translate(substr(resultfile,length(ddir)+1),'/','\')
  say ''
  return

```

Figure 77. *Car Dealer Common Interface Program*

Multimedia on the Web

Working late that day, Steve implemented a few more of the individual CGI programs. The next morning, he called Hanna and Curt over to his desk and showed them the latest additions.

“Look,” he said. “When you display the multimedia customer [*New and used cars*], you get the list of pictures, audio sounds, and videos.” [See Figure 78.]

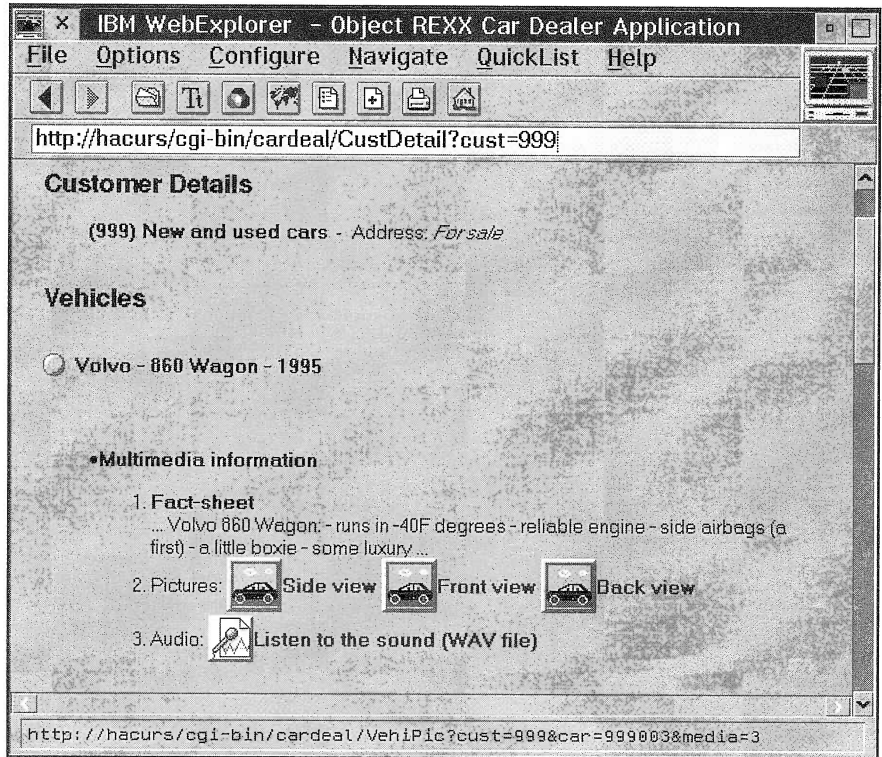


Figure 78. *New and Used Car List in Web Browser*

“Can you click on one of these to see the picture?” asked Curt.

“Yes, these are active links, and when you click on one of them, you get a new page that includes the picture of the car. Most Web browsers handle many picture formats, including BMP, GIF, and JPEG.

“When I click on the Volvo, the picture is displayed.” [See Figure 79.]

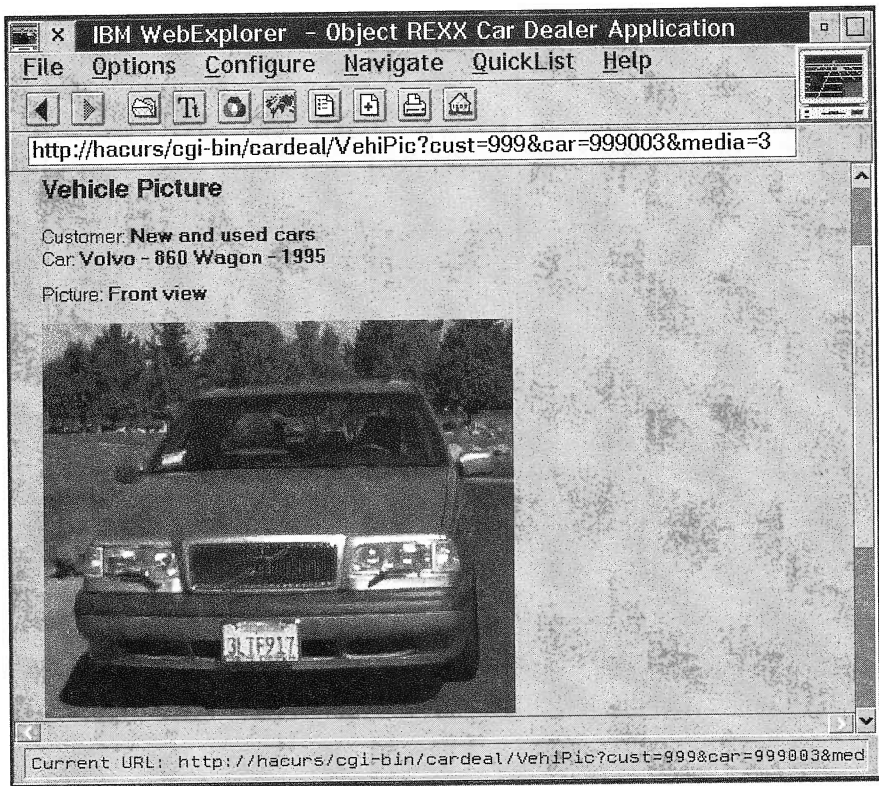


Figure 79. *Vehicle Picture in Web Browser*

"I am surprised at how fast the pictures appear," said Curt, after Steve clicked on a few more picture lines.

"Remember, we are on a local network," replied Steve. "For users on the real Internet, the pictures will appear more slowly because our BMPs are not compressed. Pictures in GIF or JPEG format are smaller than BMP, but the GUI builders do not display those formats in the GUI applications.

Try out one of the audio sounds now," he commanded Curt.

Curt clicked on an audio sound, and soon a familiar voice advertised the features of the Volvo wagon. Clicking on the simple demonstration video played the movie nicely in the multimedia TV window.

"Does every Web browser support audio and video files?" asked Hanna.

"Most browsers can be configured to invoke the operating system's multimedia function," Curt answered, before Steve even had a chance to explain how he managed to play the multimedia files on his Think-Pad.

Interacting with Web Users

One morning, Hanna arrived at the office with a new idea. She immediately called Curt and Steve over to explain her idea.

"I had a dream last night," she said. "We must involve the Web user in the application. What I want is thus: The user enters his or her name and address and information about a car. We enter this new customer and vehicle data into the database and then we let the user create a work order, select the services to be performed, and, finally, look at the bill for the job."

"That's an amazing idea," Curt shouted. "The user will come back to our home page several times to check whether the information is still there. That will prove how reliable the DB2 database is."

"What about security?" asked Steve. "The user could pretend to be somebody else when visiting our home page and create many new customers and add work orders to any of our demonstration customers. We need some control, so that each Web user can add only one car and not modify any of our own customers in the database."

"That's a real concern, Steve, you are right," said Hanna. "Maybe we can use the address field of the customer and store the Web user's TCP/IP address as a reference. Remember, the Web server passes the address in an environment variable to the CGI program," she added.

"That's a neat solution," said Steve. "Nobody will be able to touch our existing customers. But we have to extend the DB2 object model to include a method to search the customer table by address. That would enable us to check whether a customer already exists for a given TCP/IP address."

"And since we generate the resulting HTML file by the CGI program, we can include the active link to create a work order only for the customer entry of the current Web user." Curt was thinking quickly, as well.

"There is just one problem," he added. "Clever Web users can fake TCP/IP addresses and change customer records of other Web users. It's only a small problem, however, because our existing customer records cannot be touched."

"I think that's good enough for a start," said Hanna. "Our prospects are hardly of the hacker kind. Let's go to work. I will design the layout of the interactive form so that a Web user can add a customer and a vehicle. It will be a static HTML file, and I can do that!"

"Curt, you work on the program to create a new work order. You have to know how to code a CGI program; we cannot depend on Steve alone," she said, turning to Steve and smiling.

“And you, Steve, modify the existing customer display program to add an active link to create a new work order if the customer matches the TCP/IP address. And while we are at it, we can also allow Web users to delete the work order and the customer if they so choose.” she concluded.

Hanna felt good, she was in charge. It had been *her* dream, and nobody could take away *her* idea.

Adding a Web Customer

Hanna quickly designed the form for a new customer and car. She deliberately added a field for the TCP/IP address, which could then be compared with the address passed by the Web server, thus eliminating a few cases of users trying to fool the system. [See Figure 80.]

The screenshot shows an IBM WebExplorer browser window titled "IBM WebExplorer - Object REXX Car Dealer Application". The address bar displays "http://hacurs/cardeal/caryours.htm". The main content area contains a form titled "Add your own car to the database" with the instruction: "Fill in this form to add yourself and your car to the database. You can only be in the database once per TCP/IP address." The form includes several input fields: "Einstein" for Lastname, "Albert" for Firstname, "Ford" for Make, "T" for Model, "1943" for Year, and "129.33.160.66" for "Your TCP/IP address". A "Submit" button is located at the bottom of the form. The status bar at the bottom of the browser window shows "Current URL: http://hacurs/cardeal/caryours.htm".

Figure 80. *HTML Form for a New Customer and Car*

In the meantime, Steve added a *findAddress* method to the *Customer* class for both file and DB2 persistence. The new code could be tested with the file system first before running on the DB2 database. Implementing the method for both types of persistence also kept the object model in sync.

Steve then modified the customer detail page to show an active link to delete the customer and create a new work order if the address matched the TCP/IP address passed by the Web server. The additional code was simple:

```
parse arg file, type, list, who
parse var list 'cust=' custnum '?'
...
customer = custclass~findNumber(custnum)
...
if customer~address = who then
  html~~~br~href('CDDDelete?cust='customer~number, '==> Click here to delete the customer')
...
if customer~address = who then
  html~~~li~href('NewWork?cust='custnum'&car='car~serial, ,
    '==> Click here for new workorder')
```

He could always replace the active text link with a nice, small picture icon later. For now, it was important to get his code working before Curt was ready with the *customer delete* and the *create new work order* routines.

Curt implemented the delete routine with a little pain. It was his first attempt at CGI programming, and it took a few trials to get the parameters right, delete the information in the object model, and generate a suitable HTML reply.

He then tackled the new work order program. Creating a work order was simple, it just needed a customer and a vehicle; all other attributes were generated by the model. The hard part was designing the addition of service items to the work order. He decided to use an HTML form, display all the service items as check boxes, and let the user select any number of them before sending the form by using the **Submit** button. [See Figure 81.]

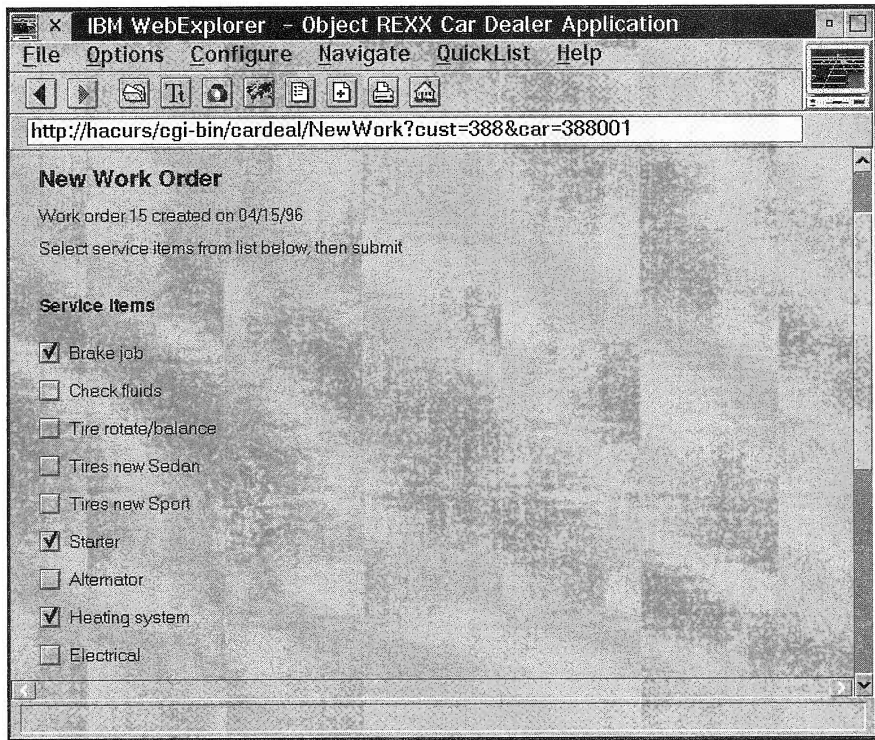


Figure 81. *HTML Form for a New Work Order*

There is a **Submit** button at the bottom of the form.

Curt decided that after processing the form the resulting Web page would be the existing customer detail display that Steve had done previously. [See Figure 74 on page 200.]

The design of the car dealer application on the Web was now complete. [Figure 82 shows the final application flow diagram.]

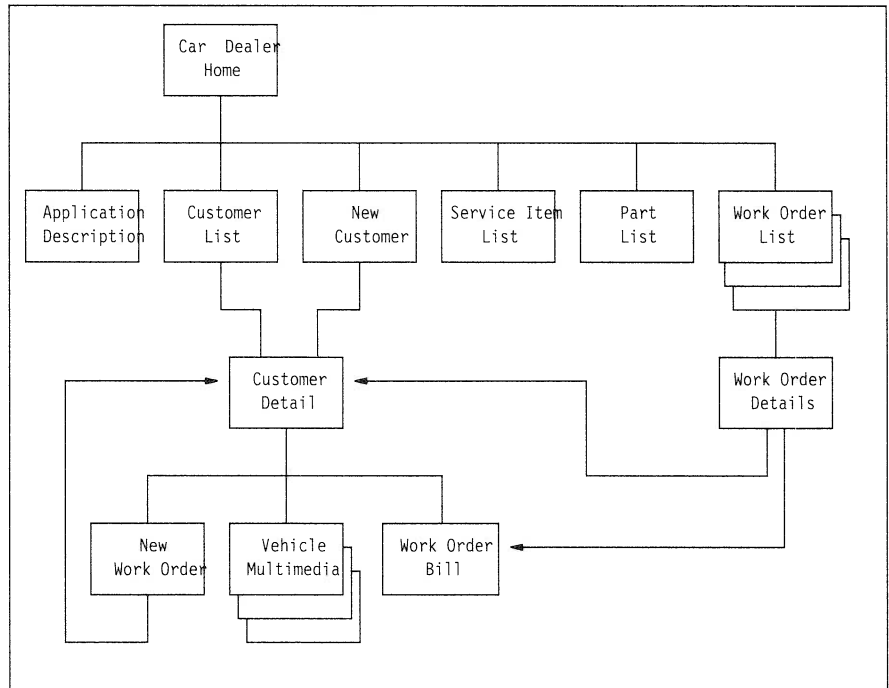


Figure 82. *Final Design for Car Dealer Application on the Web*

Car Dealer Home Page

Hanna thought about a few other pages that would enhance the function of the application, but time was running out. The leased phone line was installed, and they had to get the application out to the market.

There would be another day to make further changes. The Web was an active place, and enhancements could be added any time. Their object model and Steve's CGI program design would make it easy to maintain the attractiveness of the application and its currentness in the face of the ever-changing Web technology.

Hanna completed the car dealer home page with the new function of user interaction and tested it herself. Then she called Steve and Curt over and proudly presented the Hacurs car dealer home page [see Figure 83].

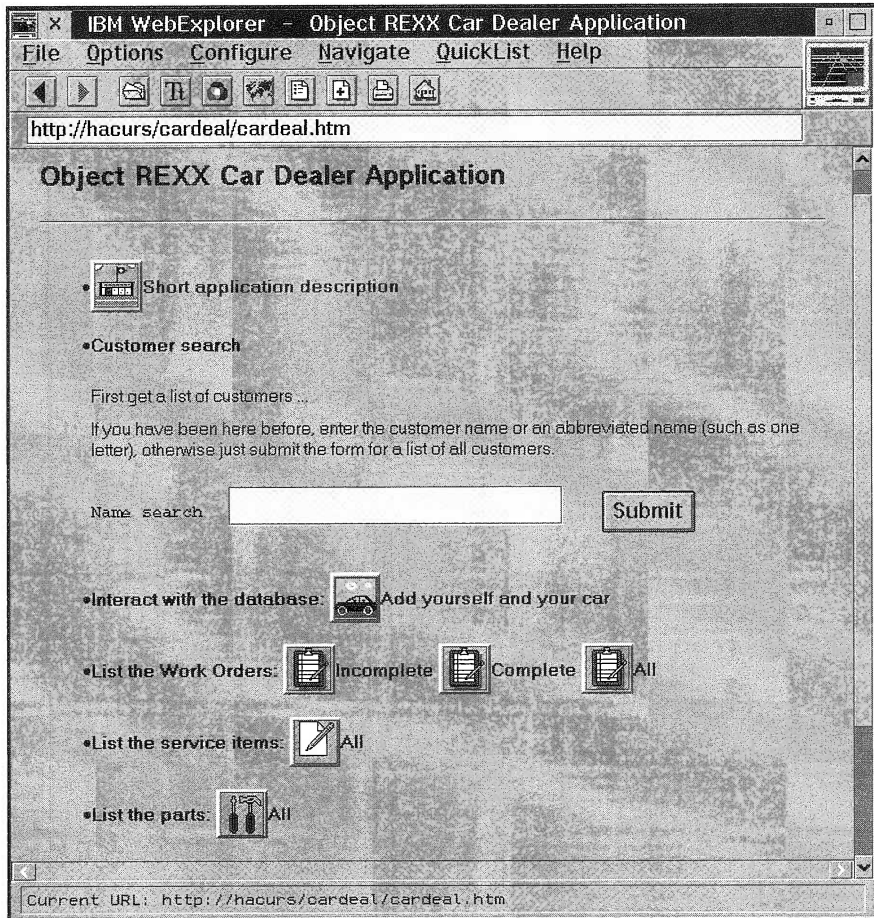


Figure 83. *Web Car Dealer Application Home Page*

"I think that's a start," said Curt, and Steve added, "You cleverly used our icons as active links to the different programs. Let's put it out on the external Internet Connection Server."

"Let the world enjoy Object REXX and the car dealer on the Web!" Hanna exclaimed, as she pushed the button to activate the external connection to their server.

Implementation Notes

Further tests showed that the car dealer application could also be started from the Web browser by invoking the *carstart* program. The *CGIREXX* interface program was enhanced with these functions:

cardeal/start?db2	Run the start program for DB2 persistence without waiting for an answer. Display the status in the Web browser.
cardeal/start?file	Run the start program for file persistence without waiting for an answer. Display the status in the Web browser.
cardeal/status	Display the status of the application in the browser.
cardeal/stop	Stop the car dealer application. Display the status in the Web browser.

These functions were implemented as methods of a class in the *carstart* program. The start method for DB2 persistence performs a logon, starts DB2, starts the application, and displays the status.

Source Code

The source code for the car dealer on the World Wide Web is not listed in the appendix; it is available in the car dealer directory on the CD or after the sample applications have been installed on a hard drive (see Table 33 on page 259).

Part 3



Object REXX and Concurrency

13

Object REXX and Concurrency

In this chapter, we experiment with the concurrency facilities of Object REXX. Object REXX provides both inter- and intraobject concurrency.

Interobject concurrency enables us to run a method against each of several different objects concurrently. Intraobject concurrency enables us to run multiple methods concurrently against a single object.

There is a detailed description of Object REXX concurrency in *Object REXX Reference for OS/2*.

Object-Based Concurrency

Every Object REXX object contains its own encapsulated method variables. It is given the processing power needed to run its methods and to exchange messages with other objects. Each object is a totally self-contained entity, and any number of objects can be active at the same time. This is defined as *interobject concurrency*. There is no danger of

multiple updates to the same object variable because each object variable is owned by only one object, and each object runs only one method at a time.

Object REXX also supports another type of concurrency, where more than one method can run against the same object at the same time. This is defined as *intraobject concurrency*. Careful planning and synchronization are needed to ensure that the variables shared between methods are updated by only one method at a time. Object REXX provides facilities to manage these aspects.

The Object REXX Concurrency Facilities

The facilities provided by Object REXX to manage concurrency are: *early reply*, *message objects*, *unguarded methods*, and the *guard* instruction.

Early Reply

A method can send an early reply to its caller using the *reply* statement, then continue running. The calling routine will be able to resume its own work while the called method continues to execute (Figure 84).

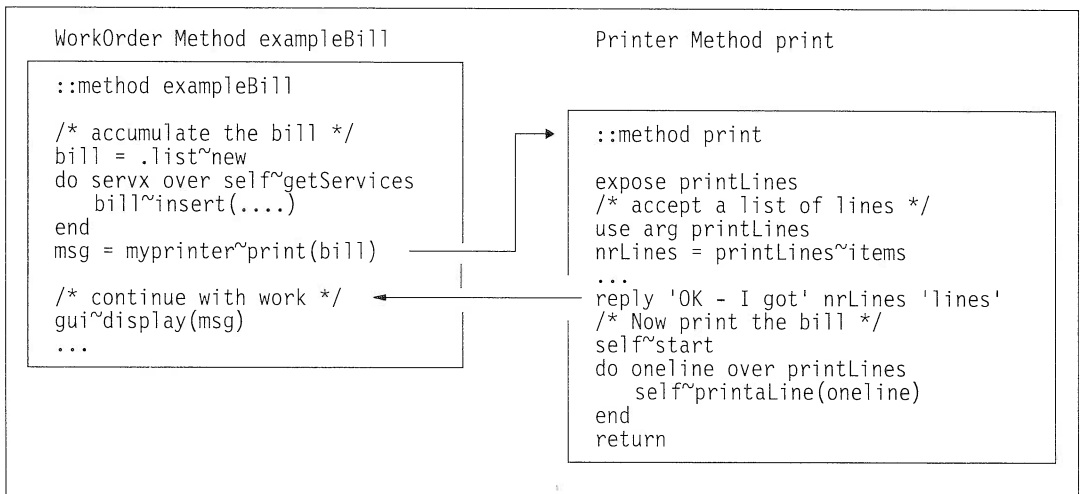


Figure 84. Concurrency with Early Reply

Message Objects

Message objects enable an Object REXX program to *start* a method executing in parallel with itself. The caller continues executing and can later ask the intermediate message object for the results of the call (see Figure 85).

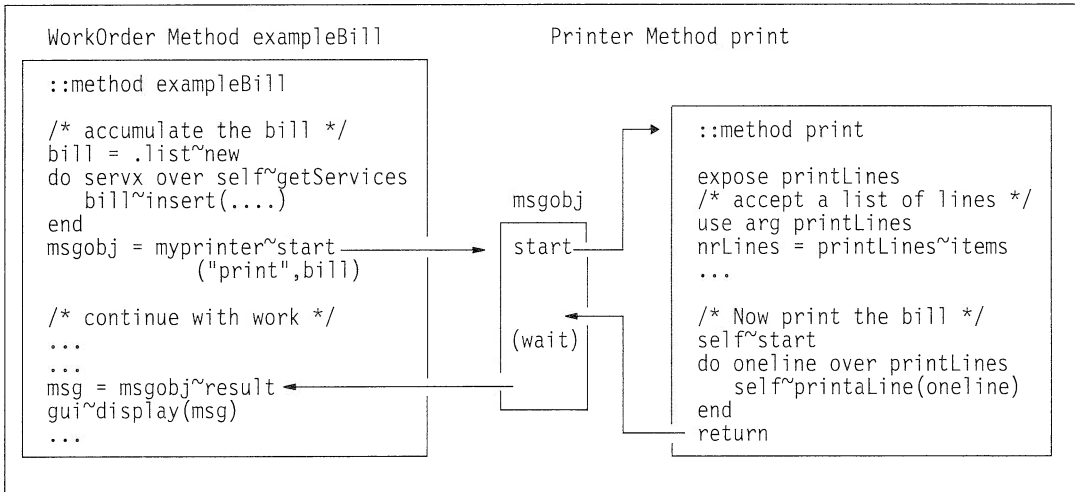


Figure 85. *Concurrency with Message Objects*

Note: When the caller asks the message object for the results, Object REXX makes it wait if the invoked method has not yet completed.

Unguarded Methods

A method can be declared *unguarded*:

```
::method getnumber unguarded
```

An unguarded method will run even if another method is already active for the same object. This enables intraobject concurrency. It is usually quite safe to make read-only methods unguarded because they do not modify the shared variable pool. It is, however, possible that some of the variables in the pool will be inconsistent with others in the same pool. Suppose, for example, that an object's methods maintain a list of numbers and the sum of all the numbers in this list within the object's variable pool. If an unguarded method reads the numbers in this list and compares their sum to the sum maintained by the other methods, the sums may differ if another method happens to be updating the list at the time it is read.

Unguarded methods are needed in recursive situations. For example, the *init* method for a vehicle invokes the *addVehicle* method of the customer, which, in turn, invokes the *getOwner* method of the vehicle to check whether the vehicle is already owned. The *getOwner* method must be declared as unguarded so that it can run in parallel with the *init* method that is already active for the vehicle in question.

The Guard Instruction

The *guard* instruction acquires or releases exclusive control of an object's variable pool. This allows a method to alternate between exclusive access to all variables and parallel execution with other methods (Figure 86).

```

/* method code */
...
guard on                /* acquire exclusive control */
                        /* wait if another method has exclusive control */
...
x = x + 1                /* process variables */
...
guard off               /* release exclusive control, allow others */
...
guard on when z > 0      /* acquire exclusive control when
                        variable z is greater than zero */
...                     /* ==> wait until z changes to positive */

```

Figure 86. *Concurrency with Guard*

For examples of the use of *guard on when* see *Coding Stored Procedures with Object REXX* on page 133 and *Philosopher's Forks* on page 223.

Examples of Early Reply with Unguarded and Guarded Methods

The example that follows shows what happens when early reply is used to achieve intraobject concurrency. We start with completely unguarded methods, which utilize full intraobject concurrency (Figure 87 on page 221).

The program contains a main routine that creates an object and sends a *repeat1* message to it. The expected result (a string) is displayed with *say*. The main routine sleeps for one second, and then displays the variable *cvar*.

If we look at the object class *example* and the three methods *repeat1*, *repeat2*, and *repeat3*, we see they are all unguarded. Thus, all three can run concurrently on the same object. The object's variables *reps* and *cvar* are concurrently available to all three methods.

The first method, *repeat1*, initializes the variable subpool with the arguments from the main routine. It immediately calls the *repeat2* method for the same object and waits.

```

/* xmpreply.cmd          Early reply example - OS/2 window          */

repetitions = 3                                /* may change          */
call RxFuncAdd 'SysSleep', 'RexxUtil', 'SysSleep'
lvar = '(Main)'                                /* init variables      */
cvar = 'Main'
cobj = .example~new                             /* allocate object     */
say lvar cobj~repeat1(repetitions, cvar)        /* - invoke repeat1   */
call SysSleep 1
say lvar 'Var =' cvar
return

::class example

::method repeat1 unguarded                      /* repeat1 method     */
  expose reps cvar
  use arg reps, cvar
  lvar = '(R1)'
  say lvar self~repeat2                        /* - invoke repeat2   */
  reply 'Reply from' lvar                     /* - early reply      */
  do reps                                     /* - loop              */
    say lvar '- Var =' cvar
    cvar = 'R1'
  end

::method repeat2 unguarded                      /* repeat2 method     */
  expose reps cvar
  lvar = '(R2)'
  say lvar self~repeat3                        /* - invoke repeat3   */
  reply 'Reply from' lvar                     /* - early reply      */
  do reps                                     /* - loop              */
    say lvar '- Var =' cvar
    cvar = 'R2'
  end

::method repeat3 unguarded                      /* repeat3 method     */
  expose reps cvar
  lvar = '(R3)'
  reply 'Reply from' lvar                     /* - early reply      */
  do reps                                     /* - loop              */
    say lvar '- Var =' cvar
    cvar = 'R3'
  end
end

```

Figure 87. *Example of Early Reply with Unguarded and Guarded Methods*

The *repeat2* method calls the *repeat3* method for the same object and waits. We now have four invocations stacked on the activity chain (the main routine, *repeat1*, *repeat2*, and *repeat3*). When the *repeat3* method issues a *reply command*, a new activity chain (thread) is started for the *repeat3* method, and control goes back to the next instruction in method, *repeat2* (the one following the invocation of method *repeat3*). Similarly, *repeat2* uses an early reply to *repeat1*, and *repeat1* uses an early reply to the main routine.

Running the program produces the kind of output shown in Figure 88. In the left column (first run) we leave all methods unguarded; in the middle column, all methods are guarded (remove the keyword *unguarded* from the source code); and in the right column, we use a mix of guarded and unguarded methods.

Methods:	UNGUARDED	GUARDED	MIXED
repeat1:	unguarded	guarded	unguarded
repeat2:	unguarded	guarded	guarded
repeat3:	unguarded	guarded	guarded
output:	(R2) Reply from (R3) (R3) - Var = Main (R1) Reply from (R2) (R3) - Var = R3 (R2) - Var = R3 (Main) Reply from (R1) (R3) - Var = R3 (R1) - Var = R3 (R2) - Var = R2 (R1) - Var = R1 (R2) - Var = R2 (R1) - Var = R1 (Main) Var = Main	(R2) Reply from (R3) (R1) Reply from (R2) (Main) Reply from (R1) (R1) - Var = Main (R1) - Var = R1 (R1) - Var = R1 (R3) - Var = R1 (R3) - Var = R3 (R3) - Var = R3 (R2) - Var = R3 (R2) - Var = R2 (R2) - Var = R2 (Main) Var = Main	(R2) Reply from (R3) (R1) Reply from (R2) (R2) - Var = Main (Main) Reply from (R1) (R2) - Var = R2 (R1) - Var = R2 (R2) - Var = R2 (R1) - Var = R1 (R3) - Var = R2 (R1) - Var = R1 (R3) - Var = R3 (R3) - Var = R3 (Main) Var = Main
Notes:	(1)	(2)	(3)

Figure 88. *Sample Output of Early Reply with Unguarded and Guarded Methods*

Notes:

1. Unguarded: The sequence in which this output appears will change each time the program is run. The three methods run in parallel and compete for processor time. Which method runs when is up to the OS/2 scheduler.
2. Guarded: Once a method gets control, it will run to completion. Only after this can another one continue. Method *repeat1* gets control first from the reply of *repeat2*, and finishes its work.
3. Mixed: Because method *repeat1* is unguarded, it can run in parallel with *repeat2*, whereas *repeat3* must wait until *repeat2* is finished.

Other combinations of guarded and unguarded methods can be tried.

Philosopher's Forks

Let's join our Hacurs team again to see a visual demonstration of Object REXX's concurrency capabilities.

On the Monday morning after a rainy weekend in October, Hanna came into the office beaming.

"Hi, Hanna, why the big smile?" called out Steve. "Are you up to something?"

"I spent the weekend playing with Object REXX's concurrency facilities. Let me show you what I built. Do you know the philosopher's forks problem?" she asked.

"Hmm," said Curt, "isn't that the one with five philosophers sitting around a table trying to grab forks and eat in turn?"

"Yes, that's the one," replied Hanna with a smile.

The philosopher's forks

- ❑ Five philosophers sit around a table. Each one goes through a cycle of sleeping and eating.
- ❑ There is a fork between each two philosophers, so there are five forks on the table, as well.
- ❑ To eat, a philosopher has to grab the forks on both sides. If a fork has already been taken by the philosopher on the other side of the fork, the philosopher must wait until that fork is free.
- ❑ The philosophers reach for forks in no particular order, but once they reach out for a fork and have to wait, they don't change their mind, even if the other fork is available.
- ❑ When they have finished eating, the philosophers put down both forks and go back to sleep.
- ❑ The times that they sleep and eat vary randomly around given values.

Philosopher's Forks in an OS/2 Window

"I wasted my weekend watching the fifth game of the World Series," said Steve. "It didn't produce a winner, and I missed the final game because I had to go to a cousin's wedding. So how did you implement the philosopher's forks, Hanna?" he asked.

"I developed a main program to control the operation and two classes—one for philosophers and one for forks," said Hanna. She opened her ThinkPad and fired it up. Steve and Curt gathered around her desk.

“The main program sets the parameters, creates the forks and philosophers, then runs all philosophers concurrently, using the *start* method,” Hanna explained [see Figure 89]. “Then it just waits for everything to finish.”

```

arg parms                                /* parameters default values: */
if parms = '' then parms = '8 6 any 2'   /* - sleep = 8 sec, eat = 6   */
parse var parms psleep peat side repeats /* - grab forks from ANY side */
T.eat = peat                             /* - run 2 cycles             */
T.sleep = psleep
T.veat = trunc(peat / 2)                  /* random variations         */
T.vsleep = trunc(psleep / 2)              /* - for eat and sleep times */
if side = 'L' then side = 100 /*left*/ /* fork side can be Left, */
else if side = 'R' then side = 0 /*right*/ /* - Right, or Any (random) */
else side = 50 /*random*/

f1 = .fork~new(1)                         /* allocate 5 forks          */
f2 = .fork~new(2)
f3 = .fork~new(3)
f4 = .fork~new(4)
f5 = .fork~new(5)

p1 = .phil~new(1,f5,f1)                   /* allocate 5 philosophers  */
p2 = .phil~new(2,f1,f2)
p3 = .phil~new(3,f2,f3)
p4 = .phil~new(4,f3,f4)
p5 = .phil~new(5,f4,f5)

m1 = p1~start("run",T.,side,repeats)     /* start the 5 philosophers */
m2 = p2~start("run",T.,side,repeats)     /* concurrently              */
m3 = p3~start("run",T.,side,repeats)
m4 = p4~start("run",T.,side,repeats)
m5 = p5~start("run",T.,side,repeats)

m1~result                                /* wait for the 5 message   */
m2~result                                /* objects to complete      */
m3~result
m4~result
m5~result
return 0

```

Figure 89. *Philosopher's Forks: Main Program*

“I’m surprised at how straightforward it looks, Hanna,” said Steve.

“The philosopher class is also quite simple,” said Hanna [see Figure 90 on page 225].

“The *init* method stores references to the fork objects and prepares an output string for indentation. The *run* method loops through sleeping and eating, picking up the forks, and laying them down again.”

“It may look simple to you, Hanna,” said Curt, “but that’s because you wrote it. Still, it is pretty short. Where’s the magic?”


```

::class phil
::method init                                /* initialization */
    expose num rfork lfork out                /* store fork objects */
    use arg num, rfork, lfork
    out = ' ' ^copies(15*num-14)              /* prepare output indentation */

::method run                                /* run through the cycle */
    expose num rfork lfork out
    use arg T., side, repeats
    x = random(1,100,time('S')*num)
    say out 'Philosopher-'num                 /* announce who you are */
    do i=1 to repeats
        stime = random(T.sleep-T.vsleep,T.sleep+T.vsleep)
        say out 'Sleep-'stime                 /* announce you are sleeping */
        rc=SysSleep(stime)                    /* sleep some random seconds */
        say out 'Wait'                        /* announce wait for forks */
        if random(1,100) < side then do        /* which fork first ? */
            lfork~pickup(1,'left',num)         /* - pick up left fork */
            rfork~pickup(2,'right',num)        /* then right */
        end
        else do
            rfork~pickup(1,'right',num)        /* - pick up right fork */
            lfork~pickup(2,'left',num)         /* then left */
        end
        etime = random(T.eat-T.veat,T.eat+T.veat)
        say out 'Eat-'etime                   /* announce you are eating */
        rc=SysSleep(etime)                    /* eat some random seconds */
        lfork~laydown(num)                    /* lay down both forks */
        rfork~laydown(num)
    end
    say out 'Done'                            /* announce you are done */
    return 1

```

Figure 90. *Philosopher's Forks: Philosopher Class*

"My secrets are hidden in the fork class," said Hanna. "That's where the concurrency and synchronization are managed, but Object REXX makes it pretty easy to do." [See Figure 91.]

```

::class fork
::method init                                /* initialization */
    expose used
    used = 0                                  /* initialize "used" flag */

::method pickup                              /* pick up the fork */
    expose used
    guard on when used = 0                    /* WAIT until "used" flag = 0 */
    used = 1                                  /* set "used" flag "occupied" */

::method laydown unguarded                   /* pay down the fork */
    expose used
    used = 0                                  /* set "used" flag to "free" */

```

Figure 91. *Philosopher's Forks: Fork Class*

"Ah," said Steve, "now it starts to get interesting. Walk us through this code, Hanna."

"The fork's *used* variable is the key," Hanna explained. "It's initially set to zero, indicating that the fork is free. The *pickup* method changes it to 1, but it contains a *guard* instruction that forces it to wait until the fork is free, which happens in the *laydown* method."

"Sounds good," said Curt, "but let's see it in action!"

Hanna started the program, and soon the window was filled with announcements of the philosophers' activities [see Figure 92].

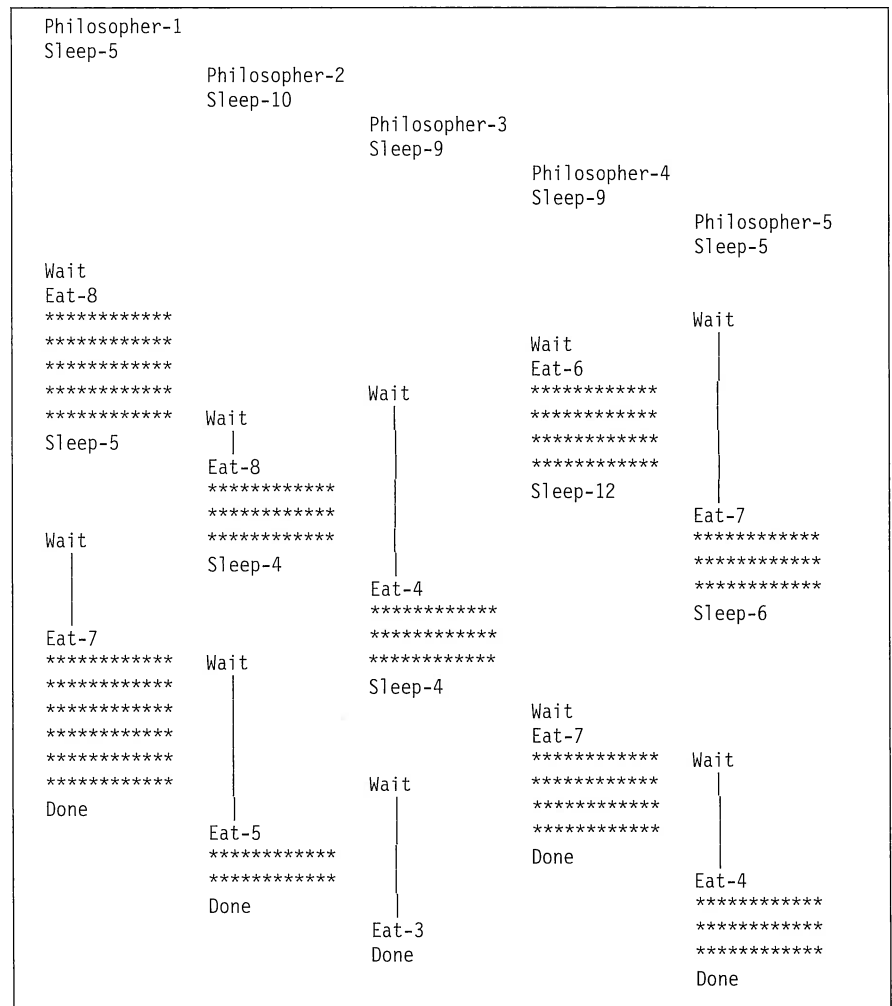


Figure 92. *Philosopher's Forks: Sample Output*

The output has been enhanced with blocks of asterisks (*) to indicate eating and vertical lines to indicate waiting. No more than two philosophers can eat at the same time because of the shared forks.

"Cool!" said Steve. "I wonder if we could use a GUI builder to make this look a bit more snazzy."

"Sounds like a great idea, Steve," said Hanna. "Why don't you try? You've got Classy Cars running so smoothly, you probably don't have anything better to do this week."

"Me and my big mouth!" said Steve with a rueful smile. "I guess I walked straight into that one. You knew that I would, didn't you? You were just waiting for me to make that suggestion," he accused Hanna. Her smile broadened, but she said nothing.

Visualizing Philosopher's Forks with a GUI

The next day, Steve came to the office late but looking rather smug. He called Hanna and Curt over to his desk to show off the colorful GUI version of the philosopher's forks. He started his ThinkPad and clicked on an icon to launch the application. A window opened and displayed Steve's inventive representation of the classical philosopher's forks problem [see Figure 93].

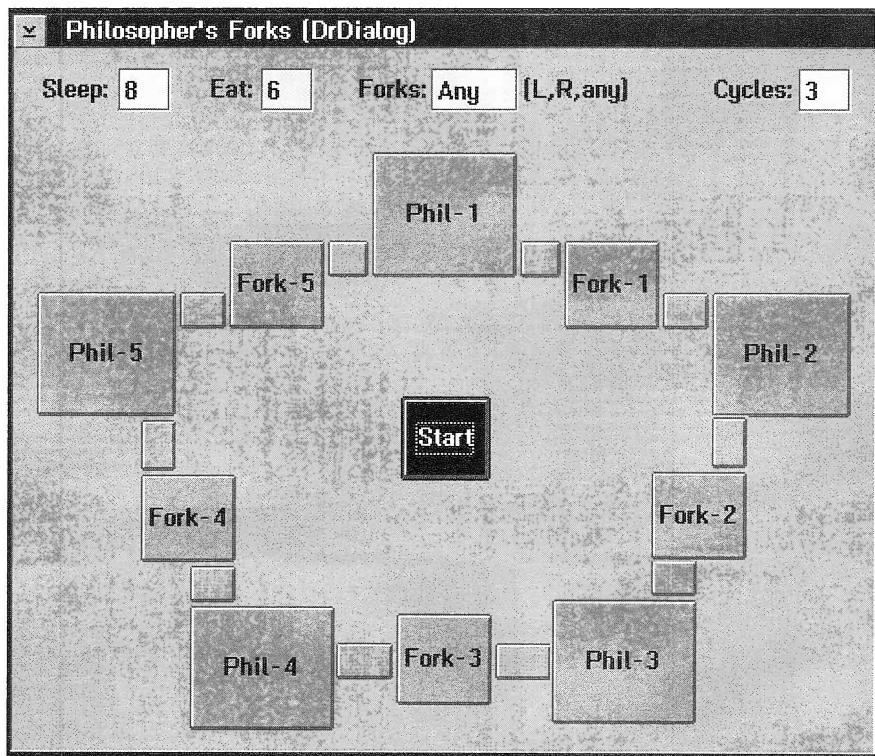


Figure 93. *Philosopher's Forks: GUI Layout*

"I implemented philosophers and forks as push buttons, and even drew little buttons between the philosophers and the forks to indicate their arms grabbing the forks," said Steve.

"That looks great, but what happens when you run it?" asked Hanna.

Steve clicked on the **Start** button, and suddenly the colors and text on the push buttons started changing. Hanna and Curt watched the unfolding story in admiration [see Figure 94].

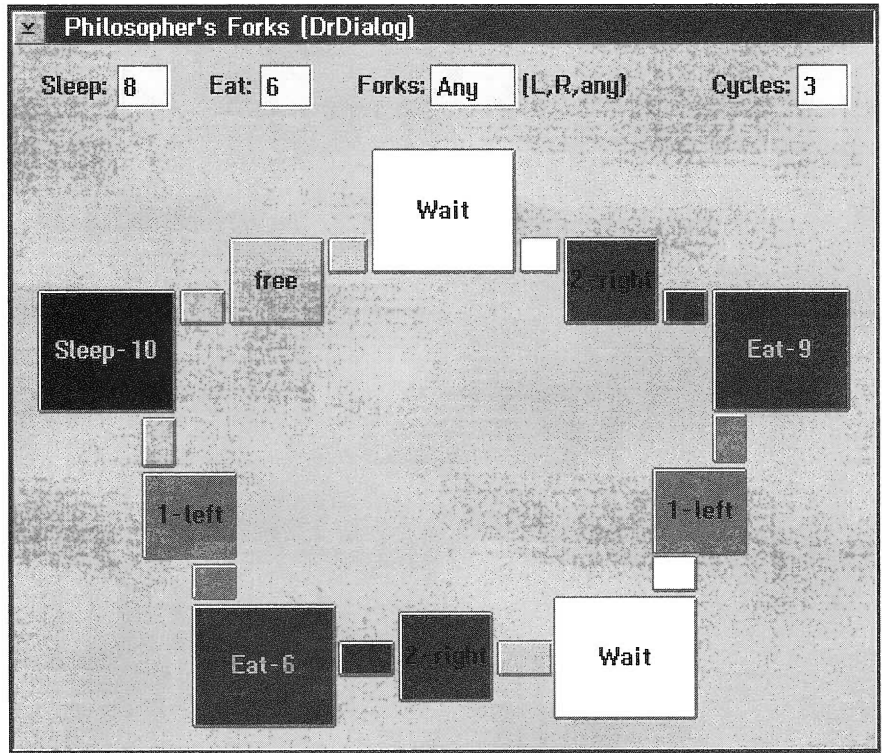


Figure 94. *Philosopher's Forks: GUI Run*

"The philosophers are black while sleeping," Steve explained.

"They turn white while waiting for a fork," said Hanna.

"And they turn red while eating," remarked Curt. "The food must be very spicy!"

"The forks also change colors," added Steve. "They turn green or blue, depending on whether they are used as a left or a right fork. They turn grey when they're not in use."

As the philosophers completed their specified number of sleeping and eating cycles, they disappeared from the screen one by one. When all had finished, the screen was reset with all the philosophers and forks back in their initial colors.

"Marvelous," exclaimed Hanna, "that looks much better than my OS/2 window version."

"Can it run any faster?" asked Curt.

"No problem," replied Steve, "I'll set the sleep and eat times to 1 second each and set it going again."

Steve did so. Now the buttons changed color much faster than before, and the three cycles completed in less than 15 seconds.

"Now let's really soup this up," said Steve. He set the times to 0 seconds and started again. Colors and text flashed rapidly across the buttons, and in just about 5 seconds it was all over.

Steve then set the number of cycles to 30 and started the application. Nothing happened. Steve turned white, as white as the five philosophers who were all waiting for a fork.

"What's happening?" asked Curt.

"It's a deadlock!" exclaimed Hanna. "Look, all the forks are green. All the philosophers happened to grab their left forks at the same time, and now they're all waiting for their right forks. How do you get out of this mess, Steve?"

Steve closed the window while he searched for a solution. "I'll have to add an interrupt button to take away the forks from the philosophers and end the deadlock," he said.

"That should do the trick," said Hanna, "and it will also allow you to interrupt the program while it's running."

"That won't take long to do," said Steve.

The three members of the Hacurs team enjoyed a hot, spicy lunch at a little Mexican restaurant near their office. Shortly after returning to his desk, Steve called Hanna and Curt over and showed them the upgraded application.

"How do you interrupt the application?" asked Hanna.

Steve just smiled and clicked on the **Start** button. It disappeared, and a red **Interrupt** button appeared in its place as the application started running.

"Sneaky!" said Hanna.

"When I click on the **Interrupt** button, the philosophers quit their cycle at the end of their current sleep or eat phase, and that makes the application stop," Steve explained, demonstrating this function as he spoke.

"Let's see if you can break a deadlock," said Curt. "Can you force one?"

"Sure," Steve responded. "I'll set the *Forks* field to *L* [left]. This will make all philosophers grab the left fork, and we'll get a deadlock."

Steve followed this procedure and was able to create a deadlock, then break it by using the **Interrupt** button.

The philosopher's forks in pictures

After we finished writing this book, we created a funny version of the application, using bitmaps, pictures, and sound. The funny application is shipped with the other samples.

The funny application is stored in subdirectory PHILFORK\ZdialFun. It uses Dr. Dialog as the GUI, but instead of colored push buttons, bitmaps of faces, hands, and forks are shown. When the philosophers are eating, they become real people, and the cake in the middle slowly disappears. In addition, one philosopher talks.

GUI Design of the Philosopher's Forks with Dr. Dialog

"Were you able to reuse the logic I developed for the OS/2 window?" asked Hanna.

"Oh yes, almost all of it," replied Steve. "I needed new code to change the color and text of all the buttons. I wanted to put as little code in the GUI builder as possible, so I put your main logic into a *Starter* class and all the GUI methods into a *GUI* class. I put the class definitions into an external file, using the same approach we took for the car dealer application."

Steve continued, "The GUI builder contains only three little pieces of code:

- [1] "Initialization when the window is opened,
- [2] "Clicking on the **Start** button, and
- [3] "Clicking on the **Interrupt** button."

Steve opened the GUI builder and showed the code to Hanna and Curt [see Figure 95].

```

Window open:  address "NULL"
               parms = '8 6 Any 3'                /* default values      */
               parse var parms sleep eat side repeats
               teat.text(eat)                      /* set entry fields    */
               tsleep.text(sleep)
               trepeat.text(repeats)
               tfork.text(side)

               .gui~initialize                      /* initialize GUI class */
               startit = .starter~new              /* prepare starter object */

Start button:  eat = teat.text()                   /* get entry fields    */
               sleep = tsleep.text()
               side = translate(left(tfork.text(),1))
               repeats = trepeat.text()

               /* run the starter object */
               smsg = startit~start("runphils",eat,sleep,side,repeats)

Interrupt button: startit~interrupt                /* interrupt method    */

```

Figure 95. *Philosopher's Forks GUI: GUI Builder Logic*

“The *Starter* class comes from your old main program, Hanna,” Steve continued. “I added an *interrupt* method to lay down all forks. I set a switch in the *local* environment, and interrogate this switch in the *interrupt* method of the philosophers.” [See Figure 96.]

```

::class starter
::method init                                /* create philosophers and forks */
  expose f1 f2 f3 f4 f5 p1 p2 p3 p4 p5
  call RxFuncAdd 'SysSleep', 'RexxUtil', 'SysSleep'
  f1 = .fork~new(1)
  f2 = .fork~new(2)
  f3 = .fork~new(3)
  f4 = .fork~new(4)
  f5 = .fork~new(5)
  p1 = .phil~new(1,f5,f1)
  p2 = .phil~new(2,f1,f2)
  p3 = .phil~new(3,f2,f3)
  p4 = .phil~new(4,f3,f4)
  p5 = .phil~new(5,f4,f5)

::method runphils                             /* start the 5 philosophers */
  expose p1 p2 p3 p4 p5
  use arg T.eat, T.sleep, side, repeats
  .gui~hideStart                               /* hide the start button */
  T.veat = trunc(T.eat / 2)
  T.vsleep = trunc(T.sleep / 2)
  if side = 'L' then side = 100 /*left*/
  else if side = 'R' then side = 0 /*right*/
  else side = 50 /*random*/
  .local [INT.FLAG] = 0                        /* initialize interrupt flag */
  do i=1 to 5
    .gui~colorFork(i,'free','free')
  end
  m1 = p1~start("run",T.,side,repeats)        /* start them here */
  m2 = p2~start("run",T.,side,repeats)
  m3 = p3~start("run",T.,side,repeats)
  m4 = p4~start("run",T.,side,repeats)
  m5 = p5~start("run",T.,side,repeats)
  m1~result
  m2~result
  m3~result
  m4~result
  m5~result                                    /* wait for finish */
  do i=1 to 5
    .gui~colorPhil(i,'black','phil','Phil-'i,'show')
    .gui~colorFork(i,'fork','Fork-'i)
  end
  .gui~showStart                               /* show start button */
  return 0

::method interrupt unguarded                  /* interrupt button clicked */
  expose f1 f2 f3 f4 f5
  .local [INT.FLAG] = 1
  f1~laydown                                  /* set the interrupt flag */
  f2~laydown                                  /* lay down all forks */
  f3~laydown
  f4~laydown
  f5~laydown

```

Figure 96. *Philosopher's Forks GUI: Starter Class*

"I took your *Philosopher* class and added calls to the *GUI* class to manage the color and text of all the buttons," said Steve. "I also added an *interrupt* method that is invoked after both eating and sleeping, to check whether the user has clicked on the **Interrupt** button." [Figure 97.]

```

::class phil

::method init
  expose num rfork lfork
  use arg num, rfork, lfork

::method run
  expose num rfork lfork
  use arg T., side, repeats
  x = random(1,100,time('S')*num)
  do i=1 to repeats
    stime = random(T.sleep-T.vsleep,T.sleep+T.vsleep)
    .gui~colorPhil(num,'yellow','sleep','Sleep-'stime)
    rc=SysSleep(stime)
    if self~interrupt then return 0
    .gui~colorPhil(num,'black','wait','Wait')
    if random(1,100) < side then do
      lfork~pickup(1,'left',num)
      rfork~pickup(2,'right',num)
    end
    else do
      rfork~pickup(1,'right',num)
      lfork~pic
    end
    if self~interrupt then return 0
    etime = random(T.eat-T.veat,T.eat+T.veat)
    .gui~colorPhil(num,'yellow','eat','Eat-'etime)
    rc=SysSleep(etime)
    lfork~laydown(num)
    rfork~laydown(num)
    if self~interrupt then return 0
  end
  .gui~colorPhil(num,'black','phil','Phil-'num,'hide')
  return 1

::method interrupt
  expose num rfork lfork
  if .local[INT.FLAG] = 0 then return 0    /** check the interrupt flag **/
  lfork~laydown(num)
  rfork~laydown(num)
  .gui~colorPhil(num,'yellow','sleep','Interrupt')
  return 1

```

Figure 97. *Philosopher's Forks GUI: Philosopher Class*

"I took your *Fork* class and added a small amount of code to invoke the *GUI* class for color and text changes," Steve continued [Figure 98.]

```

::class fork

::method init
  expose num used
  use arg num
  used = 0

::method pickup
  expose num used
  use arg seq, leftright, pnum
  .gui~colorLink(num,'wait',pnum)
  guard on when used = 0
  used = 1
  .gui~colorFork(num,leftright,seq~'leftright)
  .gui~colorLink(num,leftright,pnum)

::method laydown unguarded
  expose num used
  use arg pnum
  .gui~colorFork(num,'free','free')
  if pnum \= '' then .gui~colorLink(num,'free',pnum)
  used = 0

```

Figure 98. *Philosopher's Forks GUI: Fork Class*

"The GUI class handles all the changes in color and text, and also hides and shows the **Start** and **Interrupt** buttons when needed." [Figure 99 on page 234.]

"That's a smart design, Steve," said Hanna. "By separating all the GUI logic from the model, you've made it very easy to implement the application with VisPro/REXX or Watcom VX•REXX. The only changes would be in the GUI class and the three interactions in the GUI builder."

"Hey that sounds like a great idea!" exclaimed Steve. "Why don't you try it out, Hanna? As you say, it should be really easy. And you don't have anything better to do this week, do you?"

Hanna tried to give Steve a hard glare, but her lips kept quivering into a smile. He was only returning the favor she had done him the day before.

"Well—I'm prepared to tackle VisPro/REXX if Curt will do Watcom VX•REXX," said Hanna. She and Steve turned to look expectantly at Curt.

```

::class gui

::method initialize class
  expose color.
  color.black = '#0 0 0' /* - Black fg */
  color.yellow = '#255 255 0' /* - Yellow fg */
  color.sleep = '#0 0 0' /* - Black */
  color.wait = '#255 255 255' /* - White */
  color.eat = '#255 0 0' /* - Red */
  color.free = '#200 200 200' /* - Gray */
  color.left = '#0 255 0' /* - Green */
  color.right = '#0 0 255' /* - Blue */
  color.phil = '#0 255 255' /* - Turquoise */
  color.fork = '#255 255 0' /* - Yellow */

::method colorPhil class
  expose color.
  use arg num, fcol, bcol, text, hid
  address "NULL"
  interpret "phil"num".color('+',value('color.'fcol))"
  interpret "phil"num".color('-',value('color.'bcol))"
  interpret "phil"num".text(text)"
  if hid = 'hide' then interpret "phil"num".hide()"
  if hid = 'show' then interpret "phil"num".show()"

::method colorFork class
  expose color.
  use arg num, col, text
  address "NULL"
  interpret "fork"num".color('-',value('color.'col))"
  interpret "fork"num".text(text)"

::method colorLink class
  expose color.
  use arg num, col, pnum
  address "NULL"
  if pnum = num
    then PBid = word('pf11 pf22 pf33 pf44 pf55',num)
    else PBid = word('pf21 pf32 pf43 pf54 pf15',num)
  interpret PBid".color('-',value('color.'col))"

::method showStart class
  address "NULL" interrupt.hide()
  address "NULL" start.show()

::method hideStart class
  address "NULL" start.hide()
  address "NULL" interrupt.show()

```

Figure 99. *Philosopher's Forks GUI: GUI Class for Dr. Dialog*

"Hey, why drag me into this fight?" he asked. "You two are doing so well on your own." But their gazes did not waver from his face, and after a while he said, "Look, I can't promise anything. Unlike you two loafers, I've got to chase around the marketplace and find new business opportunities. But I'll have a look at it. Just tell me what server subdirectory you've put this stuff into."

This offer was met by cheers from Hanna and Steve, and the team settled down to work.

GUI Design of the Philosopher's Forks with VisPro/REXX

The next morning, Curt came in very late, looking tired. Hanna was able to show Steve and Curt the philosopher's forks application running with a VisPro/REXX GUI. It looked identical to the Dr. Dialog version and appeared to behave exactly the same.

"I have to invoke the GUI *initialize* method with the current *window* variable," Hanna explained. "I need the window identification in the class to set color and text. And that's all there is to it. As you can see, it follows pretty much the same pattern as your Dr. Dialog logic, Steve." [Figure 100.]

"It looks great, Hanna," Steve replied. He turned to Curt. "So, Mr. Ace Salesman, did you bring in any new business last night, or did you get a chance to look at the Watcom VX•REXX version of this application?"

"As it happened, my daughter woke us at three A.M. last night, and I couldn't get back to sleep," said Curt. "I had a stab at porting the code to Watcom VX•REXX, and this is how it turned out."

```

::class gui

::method initialize class
  expose color. window
  use arg window
  color.black = 'BLACK'           /* - Black foreground */
  color.yellow = 'YELLOW'         /* - Yellow foreground */
  color.sleep = 0                 /* #0 0 0 - Black */
  color.wait = 255*65536+255*256+255 /* #255 255 255 - White */
  color.eat = 255*65536           /* #255 0 0 - Red */
  color.free = 200*65536+200*256+200 /* #200 200 200 - Gray */
  color.left = 255*256            /* #0 255 0 - Green */
  color.right = 255              /* #0 0 255 - Blue */
  color.phil = 255*256+255        /* #0 255 255 - Turquoise */
  color.fork = 255*65536+255*256  /* #255 255 0 - Yellow */

```

Figure 100. (Part 1 of 2) Philosopher's Forks GUI: GUI Class for VisPro/REXX

```

::method colorPhil class
  expose color. window
  use arg num, fcol, bcol, text, hid
  Call VpItem window, 'PHIL' || num, 'FORECOLOR', value('color.'fcol)
  Call VpItem window, 'PHIL' || num, 'BACKCOLORRGB', value('color.'bcol)
  Call VpSetItemValue window, 'PHIL' || num, text
  if hid = 'hide' then Call VpItem window, 'PHIL' || num, 'HIDE'
  if hid = 'show' then Call VpItem window, 'PHIL' || num, 'SHOW'

::method colorFork class
  expose color. window
  use arg num, col, text
  Call VpItem window, 'FORK' || num, 'BACKCOLORRGB', value('color.'col)
  Call VpSetItemValue window, 'FORK' || num, text

::method colorLink class
  expose color. window
  use arg num, col, pnum
  if pnum = num
    then PBid = word('PF11 PF22 PF33 PF44 PF55', num)
    else PBid = word('PF21 PF32 PF43 PF54 PF15', num)
  Call VpItem window, PBid, 'BACKCOLORRGB', value('color.'col)

::method showStart class
  expose window
  Call VpItem window, 'INTERRUPT', 'HIDE'
  Call VpItem window, 'START', 'SHOW'

::method hideStart class
  expose window
  Call VpItem window, 'START', 'HIDE'
  Call VpItem window, 'INTERRUPT', 'SHOW'

```

Figure 100. (Part 2 of 2) *Philosopher's Forks GUI: GUI Class for VisPro/REXX*

GUI Design of the Philosopher's Forks with Watcom VX•REXX

Curt flipped open his ThinkPad and snapped it out of its hibernation. He clicked on an icon, and the now-familiar image of the philosopher's forks GUI implementation opened on the screen. Curt took the application through its paces, and the philosophers dined and slept once again. Hanna and Steve placed their ThinkPads alongside Curt's. There were no visible differences between the three implementations of the application.

"Well done, Curt!" said Hanna. "Show us how you coded it."

Curt started up Watcom VX•REXX and opened the philosopher's forks project. "The basic logic turned out pretty much the same as yours, Steve," he said [see Figure 101].

```

::class gui

::method initialize class
  expose color. guiname.
  use arg guiname.
  color.black = "(0,0,0)"          /* - Black fg */
  color.yellow = "(255,255,0)"     /* - Yellow fg */
  color.sleep = "(0,0,0)"          /* - Black */
  color.wait = "(255,255,255)"     /* - White */
  color.eat = "(255,0,0)"          /* - Red */
  color.free = "(200,200,200)"     /* - Gray */
  color.left = "(0,255,0)"         /* - Green */
  color.right = "(0,0,255)"        /* - Blue */
  color.phil = "(0,255,255)"       /* - Turquoise */
  color.fork = "(255,255,0)"       /* - Yellow */

::method colorPhil class
  expose color. guiname.
  use arg num, fcol, bcol, text, hid
  Call VRSet value('guiname.PB_PHIL'num), "ForeColor", value('color.'fcol), ,
                                     "BackColor", value('color.'bcol), ,
                                     "Caption", text
  if hid = 'hide' then Call VRSet value('guiname.PB_PHIL'num), "Visible", 0
  if hid = 'show' then Call VRSet value('guiname.PB_PHIL'num), "Visible", 1

::method colorFork class
  expose color. guiname.
  use arg num, col, text
  Call VRSet value('guiname.PB_FORK'num), "BackColor", value('color.'col), ,
                                     "Caption", text

::method colorLink class
  expose color. guiname.
  use arg num, col, pnum
  if pnum = num
    then PBid = word('PB_11 PB_22 PB_33 PB_44 PB_55',num)
    else PBid = word('PB_21 PB_32 PB_43 PB_54 PB_15',num)
  Call VRSet value('guiname.'PBid), "BackColor", value('color.'col)

::method showStart class
  expose guiname.
  Call VRSet value('guiname.'PB_INTERRUPT), "Visible", 0
  Call VRSet value('guiname.'PB_START), "Visible", 1

::method hideStart class
  expose guiname.
  Call VRSet value('guiname.'PB_START), "Visible", 0
  Call VRSet value('guiname.'PB_INTERRUPT), "Visible", 1
  return

```

Figure 101. *Philosopher's Forks GUI: GUI Class for Watcom VX•REXX*

“But then I hit a snag,” Curt continued. “Every object in Watcom VX•REXX has a *name* property, which is assigned a unique value when it’s created. These object names are global within a file but are hidden between files, so it’s impossible to use them between different Object REXX concurrent activities. But in addition to the name property, every object has a unique internal name assigned when it’s created. This isn’t available at design time because the internal name of an object is assigned only at run time. But the internal name can be obtained through the *self* property of the object. To be able to pick up the internal names and use them, I had to store all the GUI names into a stem (*guiname.*), which I passed to the GUI *initialize* method”:

```
gui_names = .array~of(PB_11,PB_21,PB_22,PB_32,PB_33,PB_43,PB_44,PB_54,PB_55,PB_15, ,
                    PB_Phil1,PB_Phil2,PB_Phil3,PB_Phil4,PB_Phil5,PB_Interrupt, ,
                    PB_Fork1,PB_Fork2,PB_Fork3,PB_Fork4,PB_Fork5,PB_Start)
do iname over gui_names
    guiname.iname = VRGet(iname,"Self")
end
.gui~initialize(guiname.)
```

“Great job, Curt,” said Steve. “If you’d been a bit quicker to volunteer, you might have had an easier job on your hands. But this implementation looks pretty neat to me.”

“It looks like we’ve all come up with the goods,” said Hanna. “As a reward, why don’t we treat ourselves to a good lunch?”

“That’s a great idea!” said Steve and Curt. They turned on the voice mail system, closed the office, and set out for their favorite restaurant.

Part 4



Installing Object REXX and the Sample Applications

14

Installing Object REXX and the Sample Applications

In this chapter, we discuss the installation of Object REXX and how to install and run the sample applications of this redbook.

Content of the CD

The CD distributed with this redbook contains:

- ❑ Object REXX, for installation on your machine
- ❑ The sample applications, ready to run from the CD
- ❑ The sample applications, for installation on your machine

Installation of Object REXX

Object REXX must be installed on your machine to run any of the sample applications. Run the *INSTALL* program in the CD directory OBJREXX and follow the instructions of the installation program.

Make Object REXX the default REXX by rebooting the machine, then proceed with the sample applications.

Object REXX provides the *SWITCHRX* command to switch between classic REXX and Object REXX. Reboot is necessary after each switch.

Running the Sample Applications from the CD

The CD directories CARDEAL and PHILFORK contain an executable version of the sample applications.

Use the *CAR-RUN* command in the CARDEAL directory to start any of the car dealer applications, or run the *PHILFORK* programs in any of the subdirectories of PHILFORK.

Alternatively, start the *RED-RUN* program in the CARDEAL directory, then play with the sample applications as described in *Running the Sample Applications* on page 253.

Note: You can run the car dealer application only with FAT persistence and without the *SOM Part* class. We strongly recommend to install the sample applications on your machine and experiment with the DB2 version of the car dealer application.

Installing the Sample Applications

The sample applications are delivered on the CD in the CAR-INST directory as five .ZIP files and an installation program:

- ☐ CARDEAL.ZIP - car dealer base application
- ☐ CARMED1.ZIP - multimedia data (factsheets, video)
- ☐ CARMED2.ZIP - multimedia data (bitmaps)
- ☐ CARMED3.ZIP - multimedia data (audio)
- ☐ PHILFORK.ZIP - philosopher's forks application
- ☐ INSTALL.CMD - installation command—start here
- ☐ RED_INST.EXE - installation program
- ☐ UNZIP32.EXE - unzip program
- ☐ READ.ME - latest information and pointers to the Internet
- ☐ VROBJ.DLL - needed to run VX-REXX applications

Prerequisites for the Sample Applications

The sample program requires the following:

- ☐ The system must run under OS/2 Warp.
- ☐ Object REXX must be installed and run as the default REXX.
- ☐ DB2/2 Version 2 must be installed to use the BLOBs for multimedia data.
- ☐ DB2/2 Version 1 must be minimally installed to run the application with DB2.
- ☐ OS/2 Multimedia support must be installed for audio and video play; without it, only the color pictures of the cars can be seen.
- ☐ Dr. Dialog must be installed to modify or test the Dr. Dialog applications; it is not needed for execution only.
- ☐ VisPro/REXX Version 2.1 or 3.0 must be installed to modify or test the VisPro/REXX applications; it is not needed for execution only.
- ☐ Watcom VX-REXX Version 2.1 must be installed to modify or test the Watcom VX-REXX applications; for execution only the `vrobj.dll` file from the installation directory is needed.
- ☐ SOM Toolkit 2.1 must be installed to run the SOM compiler for the part class in SOM; it is not needed for execution only.
- ☐ CSet++ or VisualAge C++ is required to compile the SOM Part class; it is not needed for execution only.

Installation Program

Start the installation using the *INSTALL* command.

The installation program preforms a number of pre-installation tasks on your system:

- ☐ Log on to the system, by default as user *userid*
- ☐ Interrogate the `OS2.INI` file for a previous installation of the sample application
- ☐ Check Object REXX WPS registration, run *WPSINST* if needed
- ☐ Check whether the redbook WPS folder already exists
- ☐ Check whether DB2/2 is installed on your system and which Version it is (only Version 2 can store BLOBs)
- ☐ Start DB2/2 and check whether the *DEALERDB* database exists

Finally, the installation menu is displayed with check marks for items that have not yet been done (see Figure 102).

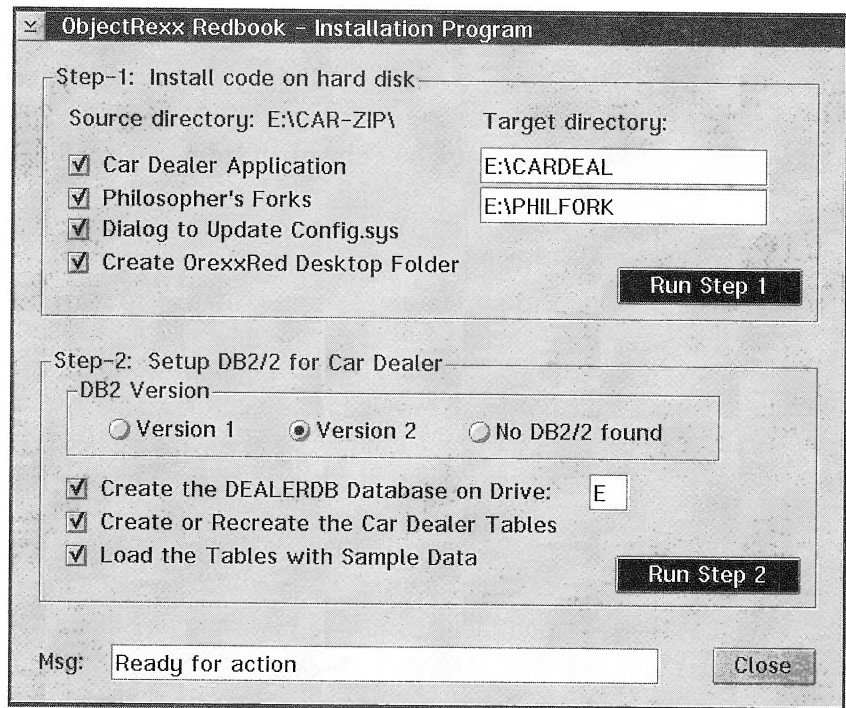


Figure 102. *Installation Program: User Interface*

The installation program delivered with the code consists of two distinct steps:

1. Installation of the code, including update of the environment and preparation of a folder
2. Setup of DB2 for persistent storage

Choose the target directories for the two applications. Use the check buttons to run only selected portions of the install program. When ready, click on the **Run Step 1** push button.

The target directories are recorded in OS2.INI as application *OREXXRED* and redisplayed automatically when the install program is restarted. A command file, sysini.cmd, is provided in the Install subdirectory to display and delete this information.

Installation of the Code

Step 1 of the installation consists of:

- ☐ Installation of the car dealer application
- ☐ Installation of the philosopher's forks application
- ☐ Updating of config.sys
- ☐ Creation of a folder with icons for all applications

Installing the Car Dealer and Philosopher's Forks Applications

The code is delivered as .ZIP files, which are unzipped into the target directories in the first two steps. A progress panel is shown while unzipping the code (Figure 103).

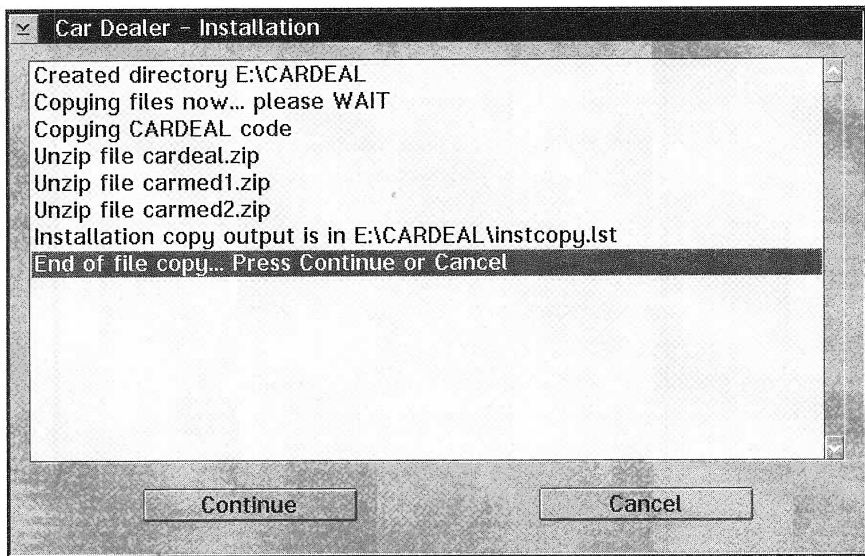


Figure 103. *Installation Program: Progress Window*

A similar progress window is displayed when installing the philosopher's forks application.

Update of Config.Sys

Config.sys will be updated by the installation program, affecting two lines:

- ☐ *SET PATH* has the CARDEAL directory added. This is necessary so that Object REXX finds the code when testing from a GUI builder.
- ☐ *SET SOMIR* has the SOM.IR file of the CARDEAL directory added. This is necessary when running the application with the *SOM Part* class.

Rebooting is necessary only before running the car dealer application from a directory other than the CARDEAL directory—for example, when testing with a GUI builder.

The changes to config.sys are shown in Figure 104.

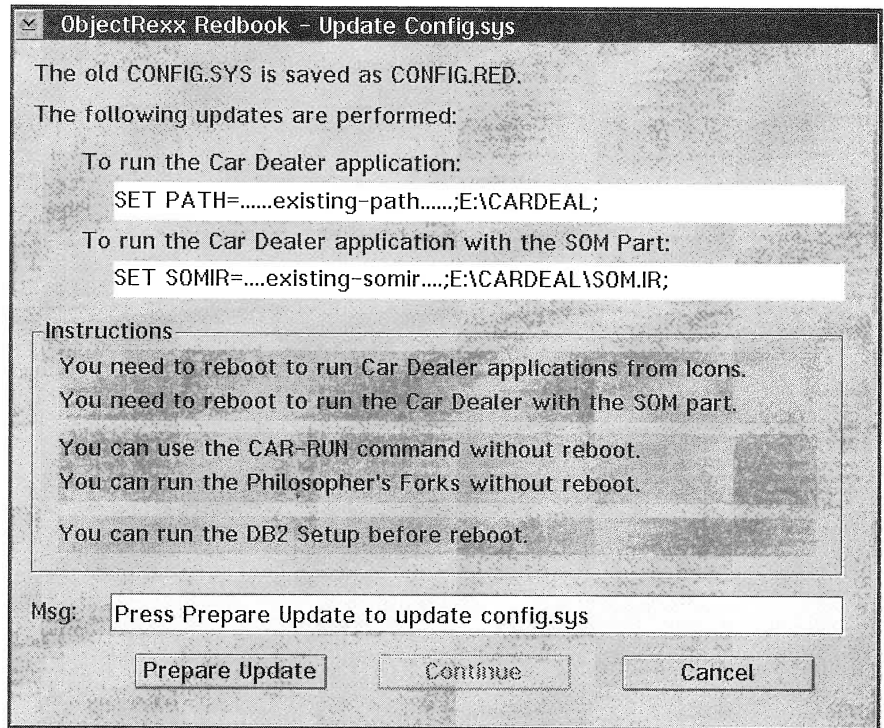


Figure 104. *Installation Program: Config.Sys Update*

Click on the **Prepare Update** push button and the changed config.sys will be displayed in a window. The actual update on the hard disk must be confirmed.

Create the ObjectRexx Redbook Folder

The last step of Phase 1 creates a nice *ObjectRexx Redbook* folder with icons for all applications and setup programs. While the folder is being created, a progress window is shown (Figure 105).

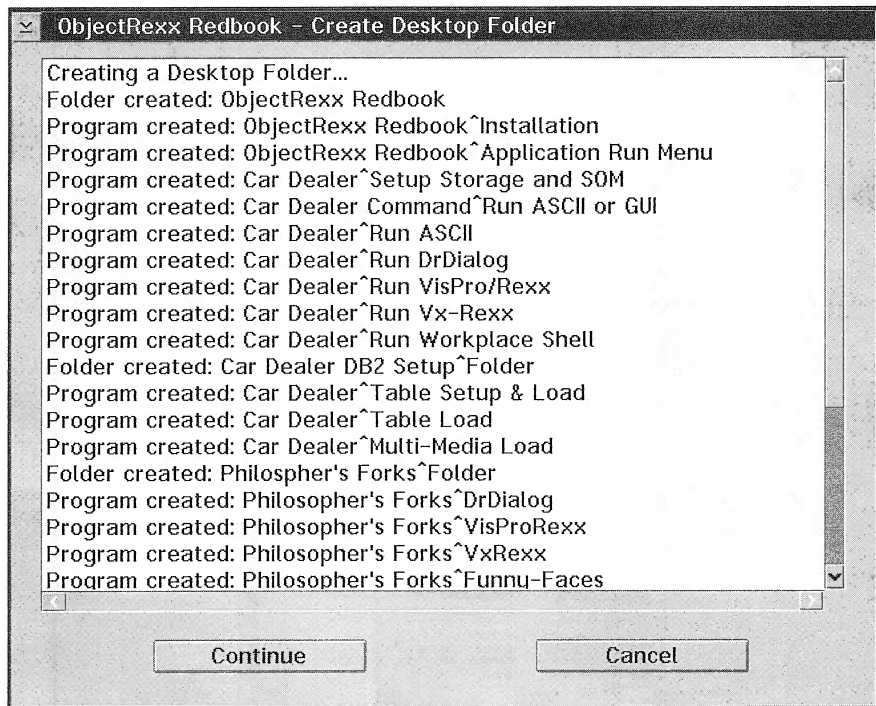


Figure 105. *Installation Program: Folder Creation*

Figure 106 shows the *ObjectRexx Redbook* folder as installed, and Figure 107 shows the *Philosopher's Forks* folder when opened.

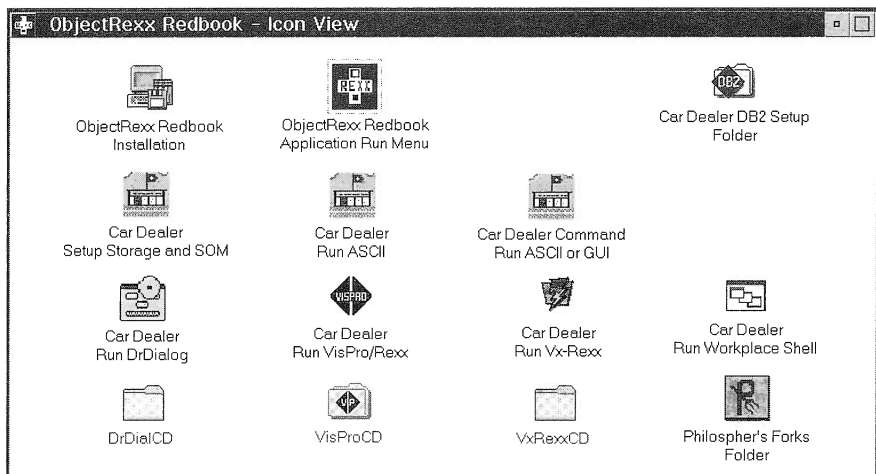


Figure 106. *ObjectRexx Redbook Folder*

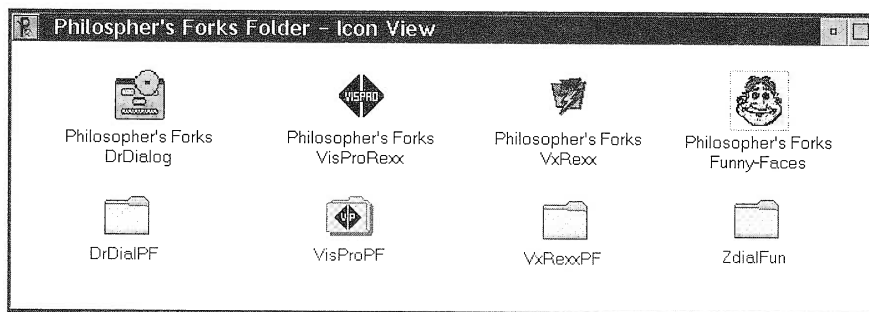


Figure 107. *Philosopher's Forks Folder*

Table 18 shows icons that are available in the two folders.

Table 18. (Part 1 of 3) Icons of the ObjectRexx Redbook Folder	
Icon	Description
 ObjectRexx Redbook Installation	ObjectRexx Redbook Installation program. Use this icon to redo certain steps of the installation, such as recreating the folder and redefining and reloading the DB2 tables.
 ObjectRexx Redbook Application Run Menu	ObjectRexx Redbook Application Run Menu (see Figure 110 on page 253). This icon permits access to all the sample programs.
 Car Dealer DB2 Setup Folder	Car Dealer DB2 Setup Folder with icons to rerun table definition and load programs. The DB2 implementation programs can run outside the installation program using the icons in this folder.
 Car Dealer Setup Storage and SOM	Car Dealer Setup Storage and SOM. Use this icon to set up persistent storage for the car dealer application and to choose whether the Part class should run from Object REXX or SOM. The option remains in effect until changed.
 Car Dealer Run ASCII	Car Dealer Run ASCII Window Program (car-aui.cmd). Start the ASCII version in an OS/2 window.
 Car Dealer Command Run ASCII or GUI	Car Dealer Run ASCII or GUI (car-run.cmd). Start any of the car dealer programs in ASCII or GUI.

Table 18. (Part 2 of 3) Icons of the ObjectRexx Redbook Folder

Icon	Description
 Car Dealer Run DrDialog	Run Car Dealer Dr. Dialog GUI program.
 Car Dealer Run VisPro/Rexx	Run Car Dealer VisPro/REXX GUI program.
 Car Dealer Run Vx-Rexx	Run Car Dealer Watcom VX-REXX GUI program.
 Car Dealer Run Workplace Shell	Run Car Dealer Workplace Shell Demonstration (carshow.cmd). Start the WPS program that visualizes all car dealer data as WPS folders. This program runs for quite a long time.
 DrDialCD	Dr. Dialog Development Folder Shadow (Car Dealer). Access the Dr. Dialog development environment.
 VisProCD	VisPro/REXX Development Folder Shadow (Car Dealer). Access the VisPro/REXX development environment.
 VxRexxCD	Watcom VX-REXX Development Folder (Car Dealer). Access the Watcom VX-REXX development environment.
 Philosopher's Forks Folder	Philosopher's Forks Folder. This folder gives access to the philosopher's forks GUI applications.
 Philosopher's Forks DrDialog	Run Philosopher's Forks Dr. Dialog.
 Philosopher's Forks VisProRexx	Run Philosopher's Forks VisPro/REXX.
 Philosopher's Forks VxRexx	Run Philosopher's Forks Watcom VX-REXX.

Table 18. (Part 3 of 3) Icons of the ObjectRexx Redbook Folder

Icon	Description
 Philosopher's Forks Funny-Faces	Run Philosopher's Forks Funny-Face application.
 DrDialPF	Dr. Dialog Development Folder Shadow (philosopher's forks). Access the Dr. Dialog development environment.
 VisProPF	VisPro/REXX Development Folder Shadow (philosopher's forks). Access the VisPro/REXX development environment.
 VxRexxPF	Watcom VX-REXX Development Folder Shadow (philosopher's forks). Access the Watcom VX-REXX development environment.
 ZdialFun	Development Folder Shadow for the funny-face application.

Note: The shadows of the GUI development folders give direct access to the appropriate development environment if that product is installed.

DB2 Setup

Step 2 of the application prepares an existing DB2/2 system for the car dealer application. This setup is optional; the car dealer application can run purely with file persistent storage. This step can be run immediately after Step 1, or at any time later. At initialization of the installation program, DB2/2 is started and checked for its Version (1 or 2).

Click on the **Run Step 2** push button to:

- ☐ Define the DB2 database
- ☐ Define the tables in the database
- ☐ Load the tables with sample data

Define the DB2 Database

The DB2 database has the name *DEALERDB*. The disk drive for the database can be selected in the entry field. Be patient while DB2 performs this lengthy step; a progress panel is not displayed.

Define the DB2 Tables

The data is stored in seven tables, one for each of the five classes and two for the m:m relationships between the classes.

If DB2/2 Version 2 is installed, the vehicle table contains the extra BLOB column for multimedia data, and two table spaces are defined to separate the basic vehicle data from the BLOB data.

With DB2/2 Version 1, only basic data can be stored in DB2. The multimedia application is still available, running from the file system.

The DDL statements are shown in a progress window while they are executed (see Figure 108).

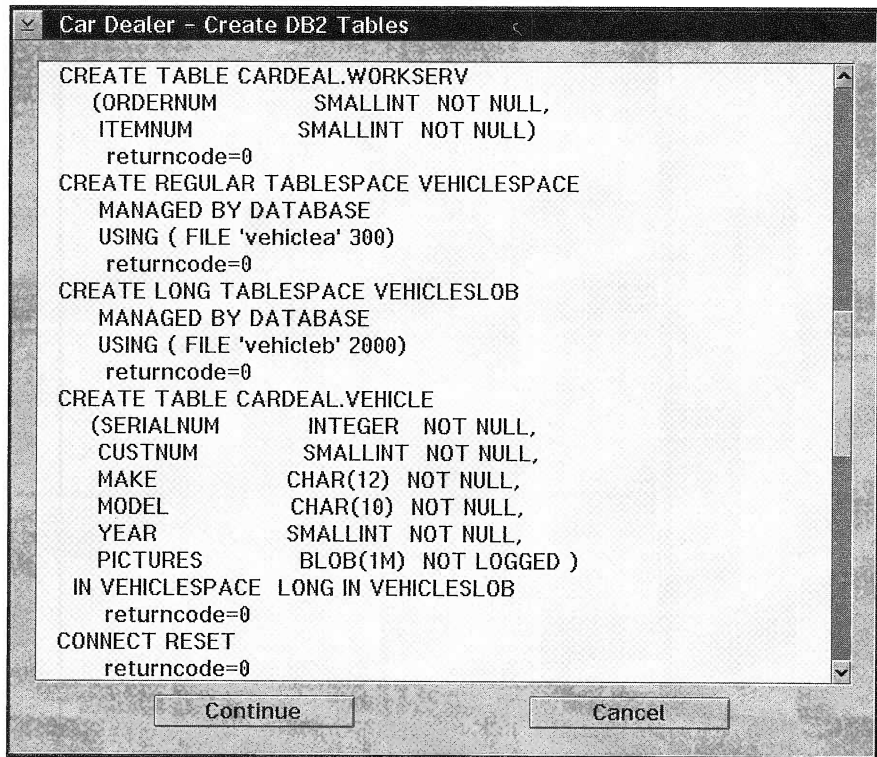


Figure 108. *Installation Program: DB2 Table Definition*

This step can be rerun at any time to redefine the tables—for example, when upgrading DB2 to Version 2.

Load the DB2 Tables

The tables are loaded using a load program that reads the sample data provided in the SampData directory. For DB2/2 Version 2, multimedia data is loaded in addition into the BLOB column. A progress window is shown while loading the tables (see Figure 109).

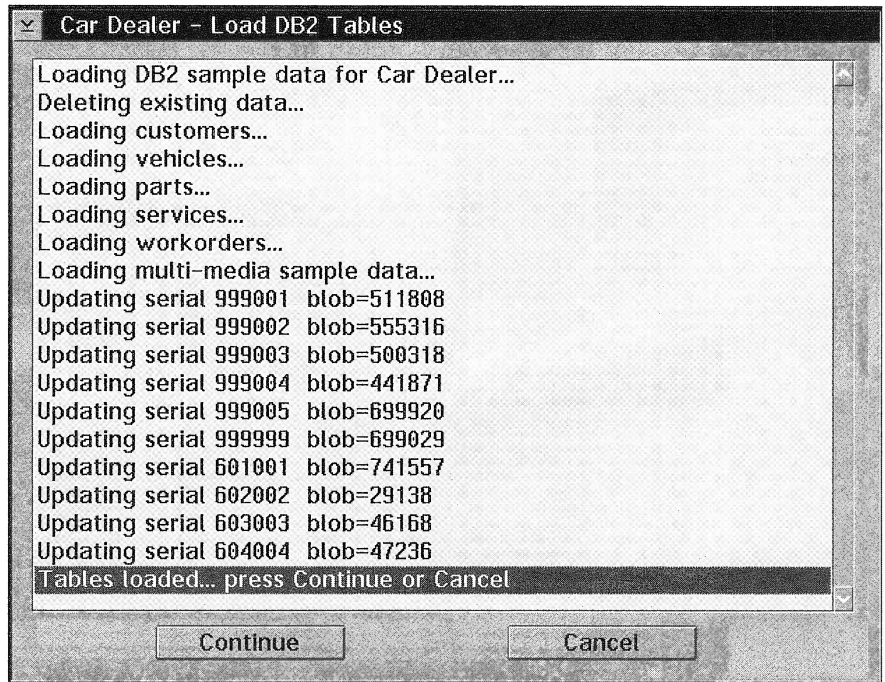


Figure 109. *Installation Program: DB2 Table Load*

This step can be rerun at any time to reinitialize the tables with the original data.

Running the Sample Applications

A menu program to run the sample applications is distributed with the code (icon **ObjectRexx Redbook Application Run Menu**). When the program is started, the window shown in Figure 110 is displayed.

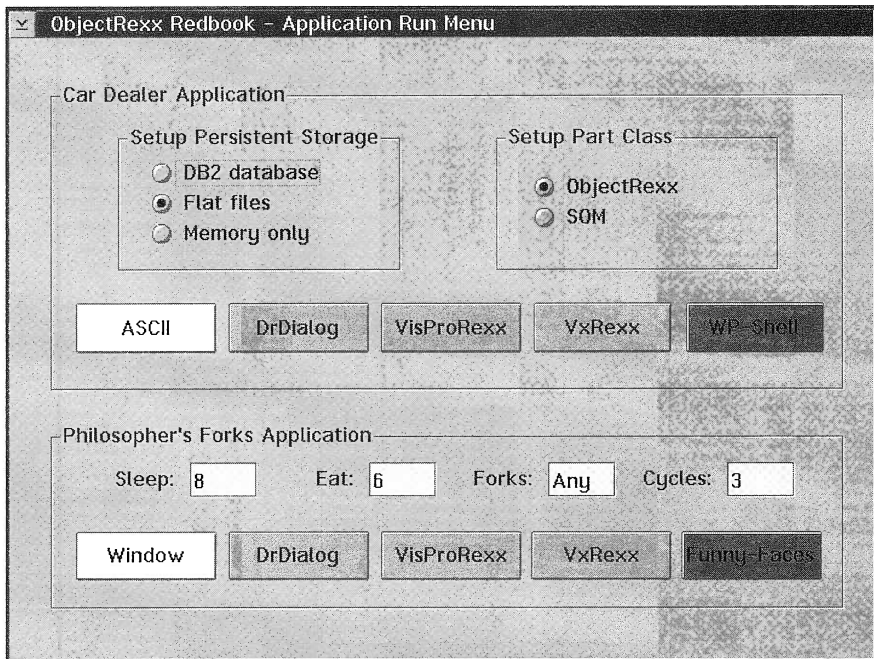


Figure 110. *Running the Sample Applications*

From this menu, all four versions of the car dealer application, the sample Workplace Shell folder demonstration program, and all five versions of the philosopher's forks can be run.

For the car dealer application, select the persistent storage and, optionally, the SOM *Part* class.

For the philosopher's forks, select the initial values of the sleep and eat times, forks sequence, and number of cycles, then invoke any one of the programs with those values.

Have fun exploring the variations of the two applications.

Running the Car Dealer Application on the World Wide Web

To run the car dealer application on the Web:

- ☐ Install the IBM Internet Connection Server on a machine with access to the CARDEAL directory.
- ☐ Tailor the configuration file (`httpd.cnf`) as described in *Customizing the File Organization on the Web Server* on page 201. The configuration must point to the CARDEAL\WWW subdirectory for HTML files and CGI programs.
- ☐ Start the car dealer application on the server in a window:


```
d:\CARDEAL\WWW\carstart db2 wait
or
d:\CARDEAL\WWW\carstart file wait
```
- ☐ Use a Web browser, for example the IBM WebExplorer, and point to the server:


```
http://hostname/cardeal/hacurs.htm
```
- ☐ Use the hot links to invoke different pieces of the application.
- ☐ When done, stop the car dealer application (press <Enter> in the window where you started it) and the server.

Installed Sample Applications

The distributed code is installed in two main directories, one for the car dealer application, and one for the philosopher's forks. By default, these directories are named CARDEAL and PHILFORK, but any name can be given.

Car Dealer Directory

Within the *CARDEAL* directory the code is structured into many sub-directories, as shown in Tables 19–35.

Table 19. (Part 1 of 2) Files of the CARDEAL Directory. Master directory of car dealer application.	
Filename	Description
<code>red-run.exe</code>	Runtime menu for applications
<code>car-run.cmd</code>	Program to run car dealer with FAT, DB2, or RAM
<code>rxftsqli.cmd</code>	Program to load REXX-DB2 functions
<code>carerror.cmd</code>	Program to check for proper directory
<code>carmodel.cfg</code>	Active configuration (copy of FAT, DB2, or RAM)
<code>som.ir</code>	SOM interface repository for the car dealer application

Table 19. (Part 2 of 2) Files of the CARDEAL Directory. Master directory of car dealer application.

Filename	Description
install.cmd	Install program to recreate database or folder
red-inst.exe	GUI installation program (called by install.cmd)

Table 20. Files of the Sampdata Subdirectory. Master files with sample data. Used as initial state for FAT persistent storage and to load the sample data into DB2 tables.

Filename	Description
customer.dat	Master file with sample customer data
vehicle.dat	Master file with sample vehicle data
workord.dat	Master file with sample work order data
service.dat	Master file with sample service item data
part.dat	Master file with sample part data

Table 21. Files of the Base Subdirectory. Base class definitions for objects in storage.

Filename	Description
carcust.cls	Base class definition for customers
carvehi.cls	Base class definition for vehicles
carwork.cls	Base class definition for work orders
carserv.cls	Base class definition for service items
carpart.cls	Base class definition for parts, a copy of part.ori or part.som
part.ori	Base class definition for parts in Object REXX
part.som	Base class definition for parts in SOM
cardeal.cls	Car dealer class for initialization and termination
persist.cls	Class for definition of persistent methods

Table 22. (Part 1 of 2) Files of the FAT Subdirectory. Class definitions for persistent objects in files.

Filename	Description
carcust.cls	FAT class definition for customers
carvehi.cls	FAT class definition for vehicles
carwork.cls	FAT class definition for work orders
carserv.cls	FAT class definition for service items
carpart.cls	FAT class definition for parts
carmodel.cfg	Configuration file for persistence in files
carlist.cfg	Configuration file for carlist.rtn (file persistence)

Table 22. (Part 2 of 2) Files of the FAT Subdirectory. Class definitions for persistent objects in files.

Filename	Description
carlist.rtn	Additional routines for list on standard output
Data	Subdirectory with persistent file storage. Initially, this is a copy of the SampData subdirectory (see Table 20). Running the car dealer application updates the files in this directory. The original state can be restored by copying the files from the SampData directory.

Table 23. Files of the DB2 Subdirectory. Class definitions for persistent objects in DB2. Initially, DB2 is loaded with data from the SampData subdirectory.

Filename	Description
carcust.cls	DB2 class definition for customers
carvehi.cls	DB2 class definition for vehicles
carwork.cls	DB2 class definition for work orders
carserv.cls	DB2 class definition for service items
carpart.cls	DB2 class definition for parts
carmodel.cfg	Configuration file for persistence in DB2
carlist.cfg	Configuration file for carlist.rtn (DB2 persistence)
carlist.rtn	Additional routines for list on standard output (DB2)

Table 24. Files of the RAM Subdirectory. Class definitions for objects in RAM. Sample data is loaded into memory using REXX statements.

Filename	Description
carcust.cls	RAM class definition for customers
carvehi.cls	RAM class definition for vehicles
carwork.cls	RAM class definition for work orders
carserv.cls	RAM class definition for service items
carpart.cls	RAM class definition for parts
carmodel.cfg	Configuration file for persistence in RAM
carlist.cfg	Configuration file for carlist.rtn (RAM, same as FAT)

Table 25. (Part 1 of 2) Files of the AUI Subdirectory. Class definitions for ASCII interface and menus, and basic list routines for displaying the class contents on standard output.

Filename	Description
caraiui.cls	AUI class with methods for window interactions
carmenu.cls	Menu class for menu display and run
menu.dat	Menu definition file

Table 25. (Part 2 of 2) Files of the AUI Subdirectory. Class definitions for ASCII interface and menus, and basic list routines for displaying the class contents on standard output.

Filename	Description
carlist.cfg	Configuration file for list on standard output; copy of same-named file from either FAT or DB2
carlist.rtn	Basic list routines
car-aii.cmd	Program to run car dealer in ASCII window

Table 26. Files of the Media Subdirectory. Media files (pictures, audio, video).

Filename	Description
media.dat	List of all multimedia files by vehicle
*.fac	Fact sheets
*.bmp	Pictures
*.wav	Audio files
*.avi	Video files

Table 27. Files of the DrDialCD Subdirectory. GUI definitions and generated executable for Dr. Dialog.

Filename	Description
car-gui.res	GUI definition file for Dr.Dialog
car-gui.rex	External routines for GUI (<i>::requires</i> carmodel.cfg)
car-gui.rxx	Generated REXX code of GUI (for listing)
car-gui.exe	Generated executable from Dr. Dialog GUI

Table 28. Files of the VisProCD Subdirectory. GUI definitions and generated executable for VisPro/REXX.

Filename	Description
car-gui.exe	Generated executable from VisPro/REXX GUI
Main	Main GUI window subdirectory
CarXxxxx	One subdirectory for each subwindow of main
SubProcs	Subdirectory for external procedures; with configuration file zCar-gui.cvp (<i>::requires</i> carmodel.cfg)

Table 29. (Part 1 of 2) Files of the VxRexxCD Subdirectory. GUI definitions and generated executable for Watcom VX-REXX.

Filename	Description
car-gui.exe	Generated executable from Watcom VX-REXX GUI
car-gui.cvx	External procedure (<i>::requires</i> carmodel.cfg)

Table 29. (Part 2 of 2) Files of the VxRexxCD Subdirectory. GUI definitions and generated executable for Watcom VxREXX.

Filename	Description
Project.VRP	Project definition file (all windows)
Window1.*	Generated REXX code (.VRM, .VRW, .VRX, .VRY)

Table 30. Files of the SOM Subdirectory. Interface definition language files to implement the part class in SOM and all files generated by the SOM and C++ compilers.

Filename	Description
part.idl	Interface definition file for part
part.xh	Generated header file
part.xih	Generated header file (updated with special code)
setpdesc.xih	File with instructions on updating part.xih
part.cpp	Generated source file (updated with own methods)
part.def	Generated DEF file for linkage editor
partmeta.idl	Interface definition file for partmeta (class methods)
partmeta.xxx	Similar to part (.xh, .xih, .cpp, .def)
parttot.def	Constructed DEF file from other DEF files
somcomp.cmd	Command file to invoke SOM compiler
complink.cmd	Command file to compile and link using C++
part.dll	Resulting DLL for LIBPATH

Table 31. Files of the StorProc Subdirectory. Sample commands to use stored procedures in a client/server environment for DB2 security purposes.

Filename	Description
server.cmd	Command file to start server for stored procedures
gateway.cmd	Command file for gateway between client and stored procedures
client.cmd	Command file for client (user of stored procedure)
read.me	A description and instructions

Table 32. Files of the WPS Subdirectory. Sample commands to visualize car dealer data in WPS folders.

Filename	Description
carshow.cmd	Program to load sample data into WPS folders
foldfind.cmd	Subroutine to search for a folder
genfold.cmd	Program to recreate Object REXX Redbook WPS folder
icons	Subdirectory with all kind of icons

Table 33. Files of the WWW Subdirectory. Car Dealer on the World Wide Web (Internet)

Filename	Description
Hacurs.htm	Hacurs home page
cardeal.htm	Car dealer main page
cardealN.htm	Car dealer application not running page
caryours.htm	Add your own car page
cardesc.htm	Short application description page
html.frm	HTML class definition
carstart.cmd	Start car dealer application (global environment)
cgirexx.cmd	Common Gateway Interface REXX program
partall*.cmd	Part list test programs 1 and 2
*.cmd	Individual CGI programs
hacurs.gif	Hacurs logo
car*.gif	Car pictures (Hanna, Curt, Steve)
*.gif	Small icon pictures for active links
http.cnf	Tailored Web server administration file (sample)

Table 34. Files of the Xamples Subdirectory. Additional small examples of the redbook.

Filename	Description
eater.cmd	Command file for eater of global object demo
feeder.cmd	Command file for feeder of global object demo
rexxcx.cmd	Command file to invoke the REXXC utility
browscls.cmd	Experimental Object REXX class browser in an OS/2 window
browser.cmd	Experimental Object REXX GUI class browser start command
browser.*	Experimental Object REXX GUI class browser Dr. Dialog files

Table 35. (Part 1 of 2) Files of the Install Subdirectory. Installation program source and executable, and DB2 setup programs.

Filename	Description
red-inst.res	Dr. Dialog GUI definition file for installation program
red-run.res	Dr. Dialog GUI definition file to run sample programs
db2setup.cmd	Program to set up and load of DB2 tables for car dealer application
load-db2.cmd	Load program for DB2 tables, uses SampData directory
load-mm.cmd	Load program for multimedia data, uses media.dat (see Table 26)
runsql.cmd	Program to run SQL DDL through DB2 command line processor
db2xmit.cmd	Program to submit DDL to DB2 from the online install program

Table 35. (Part 2 of 2) Files of the Install Subdirectory. Installation program source and executable, and DB2 setup programs.

Filename	Description
sysini.cmd	Program to display and reset OS2.INI information
createdb.ddl	DDL to create database DEALERDB
createtb.ddl	DDL to create tables for DB2 Version 2
createt1.ddl	DDL to create tables for DB2 Version 1
createix.ddl	DDL to create indexes on tables
droptb.ddl	DDL to drop tables for DB2 Version 2
dropt1.ddl	DDL to drop tables for DB2 Version 1

Philosopher's Forks Directory

Within the *PHILFORK* directory, the code is structured into only a few subdirectories, as shown in Table 36.

Table 36. Files of the PHILFORK Directory. Philosopher's Forks window and GUI programs, source, and executables.

Filename	Description
xmpreply.cmd	Sample program with early reply and unguarded methods
philfork.cmd	Philosopher's Forks in an OS/2 window
DrDialPF	Subdirectory for Dr. Dialog application
VisProPF	Subdirectory for VisPro/REXX application
VxRexxPF	Subdirectory for Watcom VX-REXX application
ZdialFun	Subdirectory for funny-faces application

Source Code for Running Sample Applications

The source code of the car dealer run program is listed in *Running the Car Dealer Application* on page 327.

Removing the Sample Applications from Your System

To remove the sample applications from your system:

- ☐ Delete the *Object REXX Redbook* folder
- ☐ Run the *Sysini* program (in the CARDEAL\Install subdirectory) to remove the sample application from OS2.INI
- ☐ Delete the CARDEAL and PHILFORK directories
- ☐ Remove the CARDEAL directory from config.sys

Part 5



New Features and Syntax in Object REXX

15

New Features in Object REXX and Migration

Object REXX is a superset of the previous OS/2 REXX. Therefore, most programs will run unchanged using Object REXX. Some small incompatibilities that may arise when migrating existing programs are discussed at the end of this chapter, in *Migration Considerations* on page 286.

Many enhancements have been built into Object REXX. The sample applications presented in this book demonstrate in detail the OO support. In this chapter, we summarize the OO support and discuss the other enhancements in detail.

Object REXX provides the following enhancements:

- ❑ A full set of OO facilities
 - Classes and methods with inheritance and polymorphism
 - A new operator, ~, to invoke methods
 - Direct access to SOM objects and the WPS
 - Concurrency—the ability to run code easily in parallel

- New special variables (*self*, *super*)
- Special and built-in objects
- ❑ A set of directives that permit
 - Definition of classes and methods (*::class* and *::method*)
 - Embedding of source files (*::requires*)
 - Creation of improved subroutines with private variables (*::routine*)
- ❑ The *REXXC* utility, which can be used to distribute programs without source
- ❑ New and enhanced instructions
- ❑ New and enhanced built-in functions
- ❑ New condition traps
- ❑ New REXX utilities

Syntax diagrams are used extensively to describe the detailed parameters of the new and enhanced instructions. The structure of the syntax diagrams is explained in Appendix B, *Definition for Syntax Diagram Structure*, on page 329.

Object-Oriented Facilities

The set of object-oriented facilities is so large that we cannot describe them all in detail here. Our intention is to add a few concepts and facilities not described in the earlier chapters of this book. We encourage study of the chapters on OO facilities in the *Object REXX Reference for OS/2*.

New Special Variables

There are two new special variables:

- | | |
|--------------|---|
| self | The object of the currently running method. Used to invoke other methods on the same object (<i>self~display</i>) or to pass as a parameter to a method of another object (<i>.Customer~addVehicle(self)</i>). |
| super | The superclass (parent in inheritance hierarchy) of the current object. Used to invoke a method in the superclass, in many cases the method of the same name. For example, in the <i>init</i> method of a class it is common to invoke the <i>init</i> method of the parent (<i>self~init:super</i>). |

Special and Built-In Objects

Object REXX provides a set of objects that are always available:

.environment

The global environment object. It contains all predefined class objects (*.Object*, *.String*, ...) and some other objects (*.true*, *.false*, *.nil*). It can be used for communication among multiple processes (see *The Global Directory* on page 154).

.nil

The *NIL* object, an object that does not contain any data. It can be used to test for nonexistent data—for example, in an array:

```
if myarray[i] = .nil then ...
```

.local

The local environment object. It contains default input/output streams (*.input*, *.output*, *.error*) and some SOM-related objects (*.som*, *.somclass*, ...). It can be used for communicating among parts of the application within one process (see *The Local Directory* on page 153).

.methods

A directory of methods defined in the current program using *::method* directives without an associated class.

.rs

The return code from any executed command, with values of -1 (failure), 1 (error), 0 (OK).

Directives

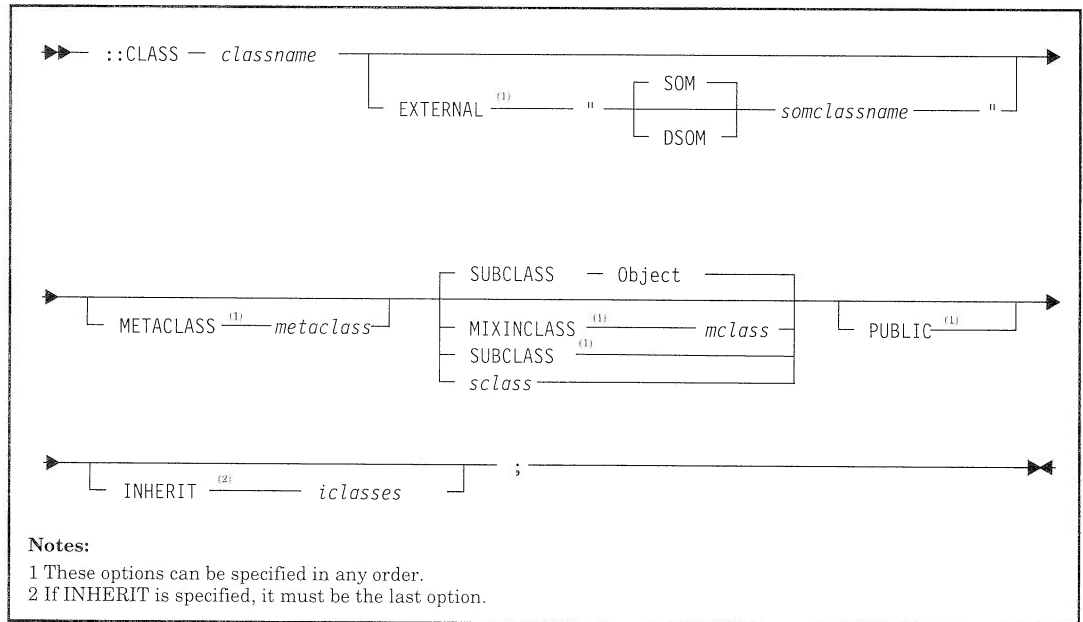
Object REXX provides four directives, two to define classes and methods, one to define external routines, and one to implement dependencies between source files.

Directives are nonexecutable and must be placed at the end of the source file. They are processed first to set up a program's classes, methods, and routines.

Class Directive

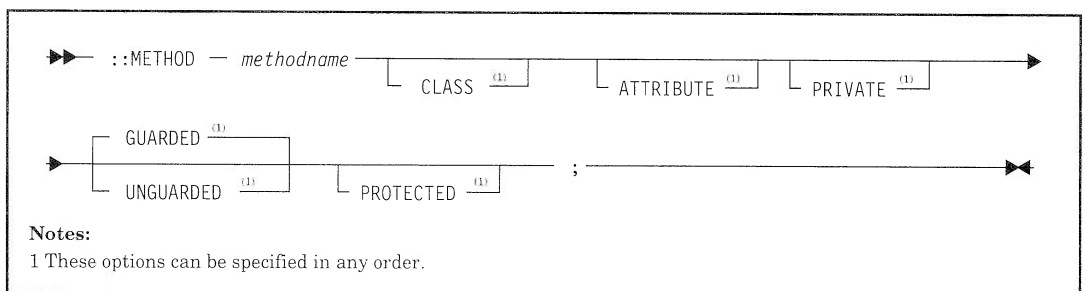
The *::class* directive defines a new class. Several options are available:

public	Makes the class available in all programs that have a <i>::requires</i> directive for this program
subclass	Inherits from a parent class
inherit	Inherits from other mixin classes
mixinclass	Defines a mixin class for inheritance
metaclass	Defines a meta class for additional class methods
external	Retrieves class from SOM interface repository



Method Directive

The `::method` directive defines a method. Multiple method directives are usually placed directly after the class directive. All options except *protected* are described in this redbook. The *protected* option deals with the Security Manager, an Object REXX feature that has not been used in this redbook.



Routine Directive

The `::routine` directive defines a callable subroutine. Such routines behave like external routines but are in the search order before external routines (after internal ones). The only option is *public*, which makes the routine available to all programs with a `::requires` directive for this program.

```

▶▶ ::ROUTINE — routinename — PUBLIC — ;

```

Requires Directive

The `::requires` directive specifies that a program requires access to another source program. In many cases, the other program contains class definitions needed for execution. The `::requires` directive allows the building of libraries of reusable code and the implementation of configuration management of REXX programs (see Chapter 10, *Configuration Management with Object REXX*, on page 143). The `::requires` directives must precede all other directives.

```

▶▶ ::REQUIRES — programname — ;

```

The REXXC Utility

The *REXXC* utility can be used to transform a source program into an executable image that can be distributed without the source code:

```
REXXC inputfile outputfile
```

When there are multiple programs that call each other, it is necessary to keep the same file names after transformation. There are basically two approaches:

- ❑ Use the same names for the output files but place them in a different matching directory structure.
- ❑ Transform the source into an output file and, when successful, save the source under a different name and rename the output to the name of the original source. (With HPFS drives, the source can be saved as *filename.ext.rxc*, for example, as implemented in the *REXXCX* command in the *Xamples* subdirectory of the car dealer application.)

New and Enhanced Instructions

The new instructions added to Object REXX are:

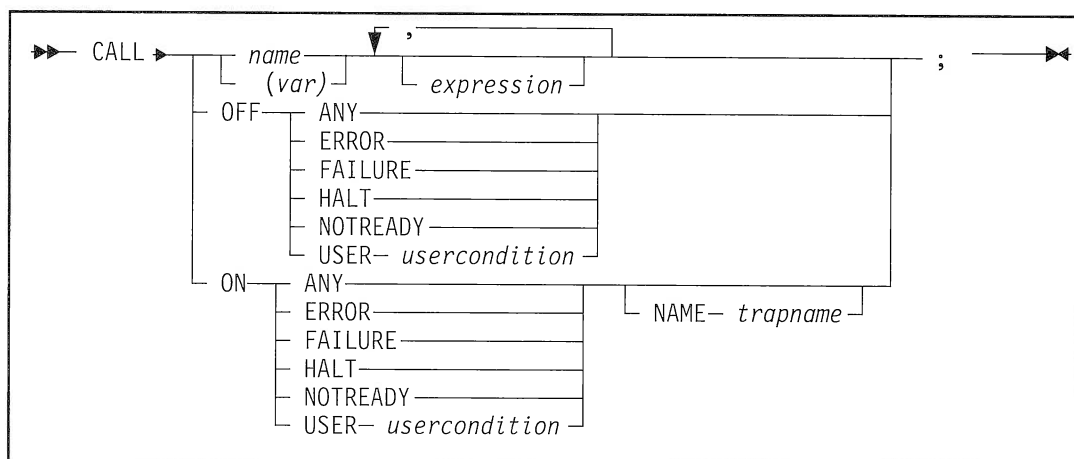
- ☐ EXPOSE
- ☐ FORWARD
- ☐ GUARD
- ☐ RAISE
- ☐ REPLY
- ☐ USE

The parameters for four old instructions have been enhanced:

- ☐ CALL
- ☐ DO
- ☐ PARSE
- ☐ SIGNAL

The new and changed instructions are discussed in alphabetical order.

CALL (Enhanced)



The first new feature on the CALL instruction is that *(var)* can now be used instead of *name* to specify the routine to be called. The variable is evaluated first, and the resulting value is used as the target of the CALL instruction. Observe that this value is not changed to upper-case, so it must exactly match the label to be called. In this small example, there are three different ways of calling internal and external routines:

```
/* TstCALL.CMD - Test of "CALL (var)" instruction */
Call label calldata      /* label is a symbol (constant) */
label = 'label'
Do 2
  Call (label) calldata  /* label is a variable */
  label = 'newlabel'    /* - that changes */
```

```
End
Call "label" calldata      /* label is a string */
exit
label:
  Say "The first call was made to label - label:"
  return
"label":
  Say 'The second call was made to label - "label":'
  return
"newlabel":
  Say 'The third call was made to label - "newlabel":'
  return
```

The last call to "label" bypasses any search for an internal routine and calls an external command file named LABEL.CMD:

```
/* LABEL.CMD - test with external routine */
Say 'The fourth call was made to external routine -
LABEL.CMD'
return
```

Running the TstCALL.CMD gave the expected result. The little *do* loop (Do 2) caused the same call statement to call two different routines. The variable *label* was evaluated correctly.

```
[C:\]TstCALL
The first call was made to label -> label:
The second call was made to label -> "label":
The third call was made to label -> "newlabel":
The fourth call was made to external routine -> LABEL.CMD
```

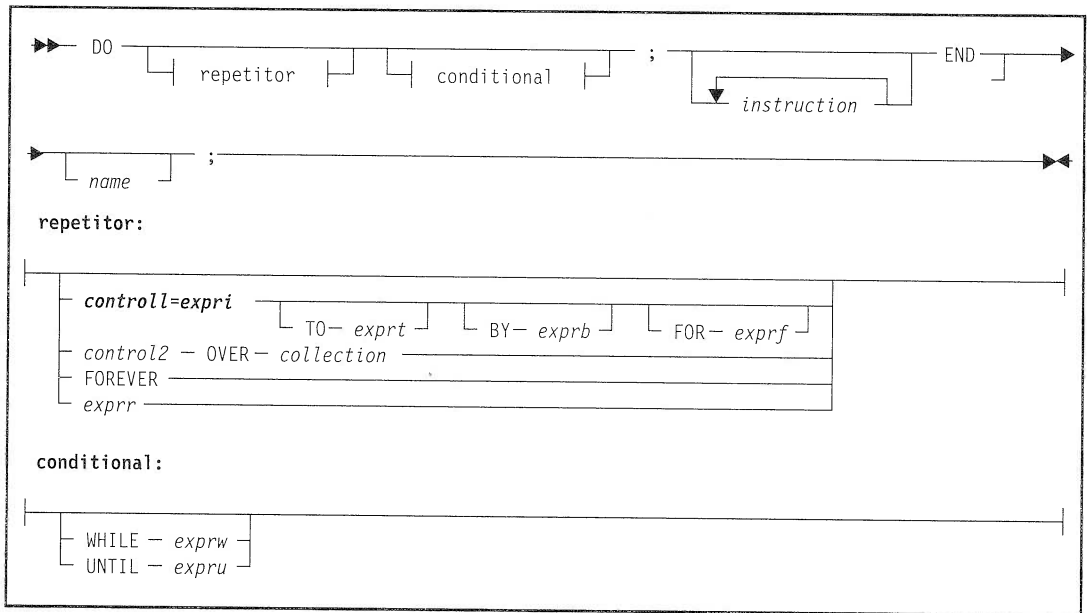
Also, the CALL instruction has two new conditions, *ANY* and *USER*, added. They are explained in *SIGNAL (Enhanced)* on page 274, and we will come back to these in connection with the rest of the new condition traps in *New Condition Traps* on page 280.

DO (Enhanced)

The DO instruction has a new repetitor function added that will make it possible to loop through all values of a stem object or any other collection that provides a *makearray* method. The repetitor is coded as *control2 OVER collection* in the syntax diagram below.

The *DO xvar OVER Stemx.* sets the variable *xvar* to each one of the member names of the *Stemx.* stem object. This is very useful because we no longer have to know the names of the tails in a stem variable. The *DO .. OVER* gives all the tails, but in any order, so please do not rely on the order.

DO OVER works very well with the collection classes of Object REXX, such as lists, arrays, sets, tables, bags, and relations. The car dealer application uses it extensively.



EXPOSE (New)

The *EXPOSE* instruction is new for Object REXX. Before, we had the *EXPOSE* option on the *PROCEDURE* instruction. The *PROCEDURE* instruction protected the variables of the calling routine. If the routine needed access to some of those variables, we used the *EXPOSE* option to make them available. The new *EXPOSE* instruction has a very similar function for the variables of an object. It is used to expose the instance or class variables of a method from the object's variable pool. The *EXPOSE* instruction can be used only in a method and, if used, it must be the first instruction after the *::method* directive.

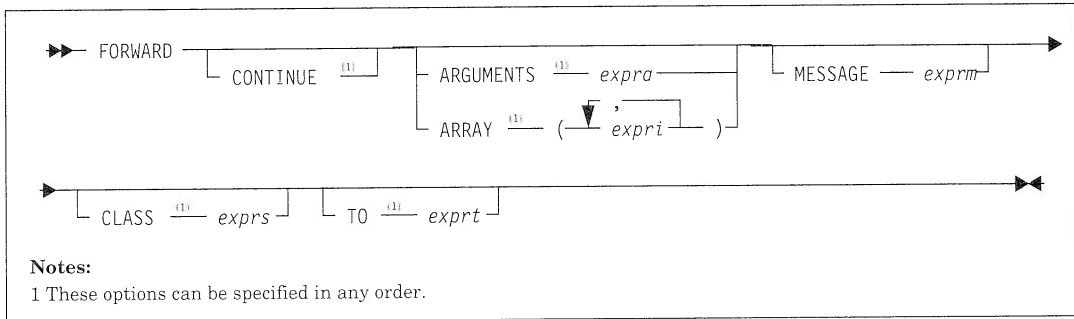


FORWARD (New)

This new instruction is used to forward a message that caused the currently active method to start running. Parts of the forwarded message can be changed by the different options on the *FORWARD* instruction. Target object, arguments, and even the message name can be changed.

One use of *FORWARD* is to pass on a message to the superclass if the current method is overriding a method of that class but still wants that method to run. The *CONTINUE* option decides whether a return

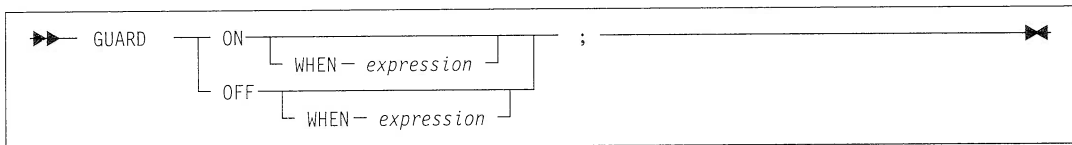
should be made to the forwarding method. It also decides how any result should be handled. The *FORWARD* instruction causes no concurrency—the forwarding method waits for the return (if *CONTINUE* is specified) or exits directly after forwarding the message.



GUARD (New)

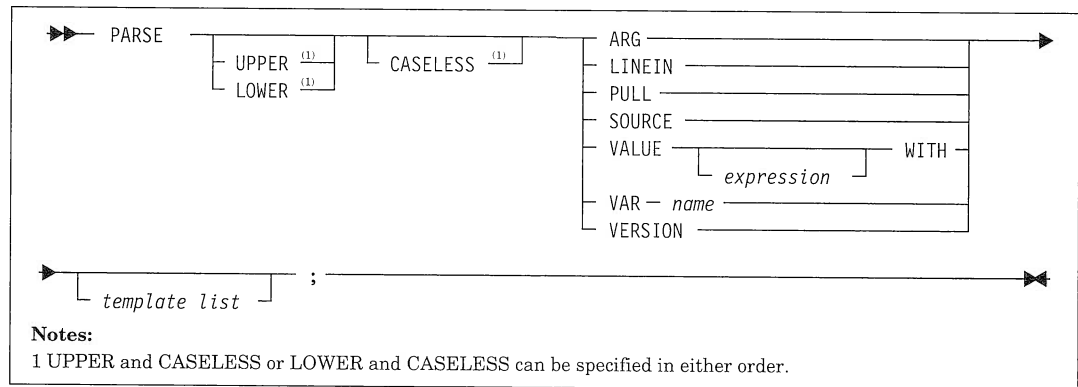
The *GUARD* instruction is used to control access to an object's variable pool. The normal state for an object is that it is guarded from concurrent use by different methods. Sometimes we want to let multiple methods share the use of one object's variable pool. This is done by using either *methodname~setunguarded* or *::method methodname unguarded*. The *GUARD* instruction can now be used to temporarily lock out concurrent use of the object's variable pool. The option *when expression* can make it conditional.

Examples of *GUARD* are used in *Coding Stored Procedures with Object REXX* on page 133 and in the fork class in the philosopher's forks (see Figure 91 on page 225).



PARSE (Enhanced)

The *PARSE* instruction has two small enhancements. The *upper* option is now complemented with a *lower* option; thus, any character string to be parsed is first translated to lowercase. The other new option—*caseless*—causes any matching done during parsing to be independent of case; a letter in uppercase is thus equal to the same letter in lowercase.



Examples:

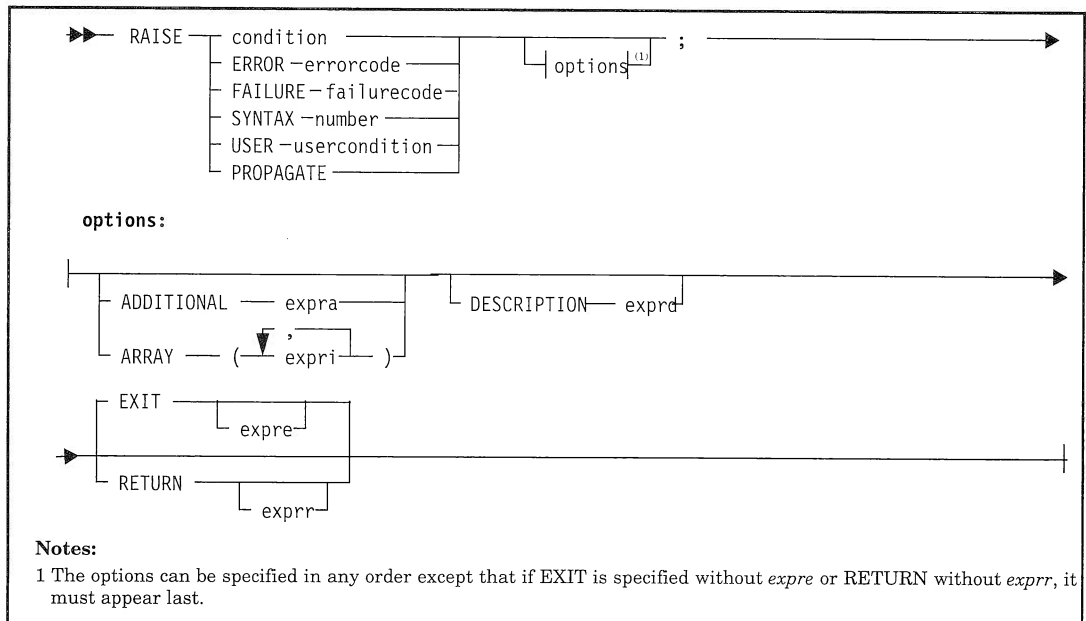
```

parse value 'AbCdEfGhIjKlM' with p1 'FgH' p2
==> p1 = 'AbCdEfGhIjKlM', p2 = ''
parse caseless value 'AbCdEfGhIjKlM' with p1 'FgH' p2
==> p1 = 'AbCdE', p2 = 'IjKlM'

```

RAISE (New)

Traps are normally created totally involuntarily. *RAISE* is a new instruction that enables the programmer to create traps in a controlled way.



One nice use of the *RAISE* instruction is to have a routine for catching condition traps for methods without having to add a lot of code to each method.

The following is an example of *raise propagate*:

```
/* TstRaise.Cmd - Test the new RAISE instruction */
signal on any
tm = .myTest~new
say tm~myMethod
exit
any:
signal off any
if .local["M.SIGL"] <> .nil then do
    sigl = .local["M.SIGL"]
    .local["M.SIGL"] = .nil
end
if var('rc')
    then say 'REXX ['condition("C")'] error' rc 'in line' sigl':' ,
            "ERRORTEXT"(rc)
    else say 'REXX ['condition("C")'] error in line' sigl
say 'The Source Line is:"SOURCELINE"(sigl)
exit
::class myTest
::method init
return
::method myMethod
signal on any
a = 'xyz'
c = a+2    /* This line causes SYNTAX error */
return
any:
.local["M.SIGL"] = sigl
raise propagate
===> Result:
REXX [SYNTAX] error 41 in line 25: Bad arithmetic conversion
The Source Line is:  c = a+2
```

REPLY (New)

REPLY is used to send an early reply from a method to the caller, removing the method from the current activity stack and letting it run concurrently with the caller. This is one of the ways to cause concurrency under Object REXX. See *Examples of Early Reply with Unguarded and Guarded Methods* on page 220. Observe that *REPLY* can be used only within methods, and it can be executed only once within a method.

►► REPLY

└ expression ┘

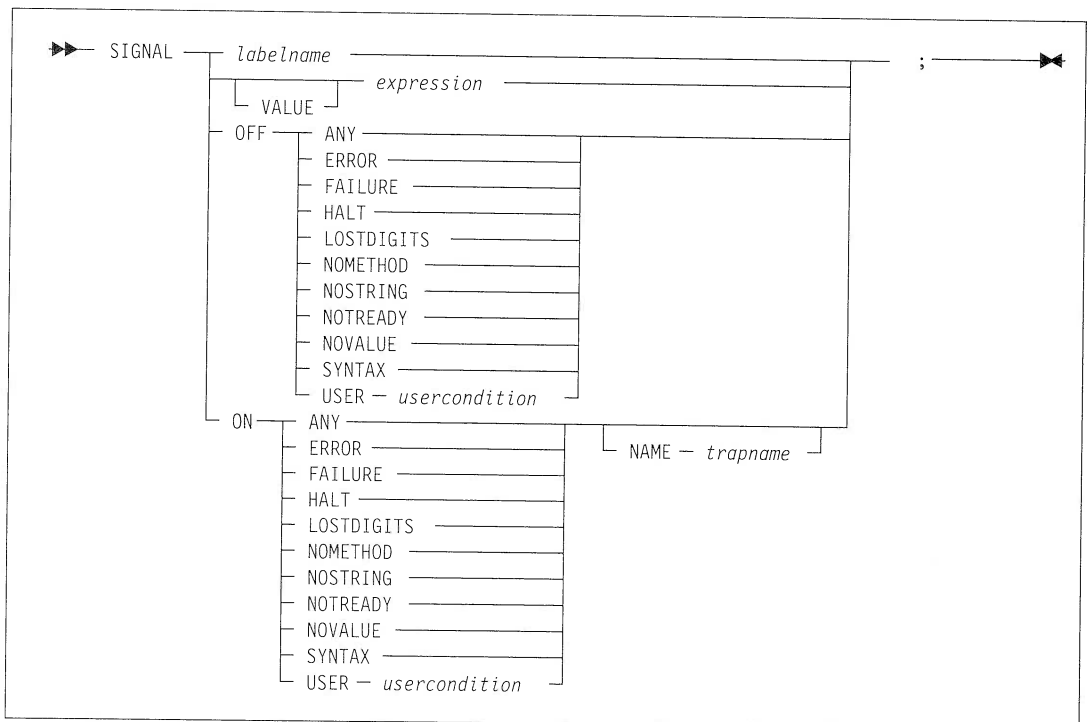
;

◄◄

SIGNAL (Enhanced)

SIGNAL is used to cause an *abnormal* change in the flow of control or, if *ON* or *OFF* is specified, it controls the trapping of specific conditions. In Object REXX, some new conditions have been added:

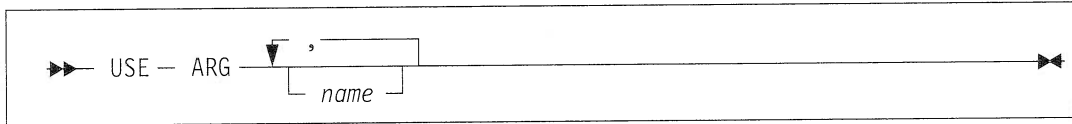
- ☐ *ANY*—traps any condition not specifically enabled by the other condition settings
- ☐ *LOSTDIGITS*—detects when a number in an arithmetic operation has more digits than the current setting of *NUMERIC DIGITS*
- ☐ *NOMETHOD*—detects when an object receives an unknown message and there is no *unknown* method to receive it
- ☐ *NOSTRING*—detects when a string value is required from an object and it is not supplied
- ☐ *USER usercondition*—allows the setup of user conditions invokable by the *RAISE* instruction that specifies the same *usercondition* name.



For more information on conditions and *SIGNAL*, see *CALL/SIGNAL (Enhanced)* on page 280.

USE (New)

USE ARG retrieves the argument objects provided to a program, routine, function, or method. The objects are assigned into variables.

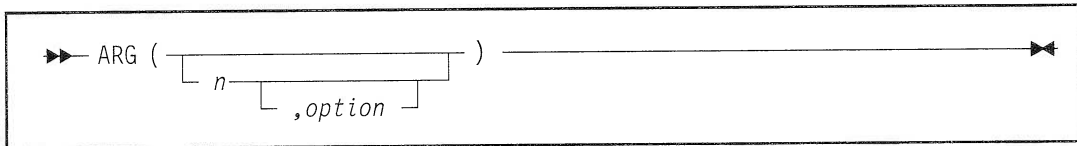


The difference between *USE ARG* and *PARSE ARG* is that *PARSE ARG* (and *ARG*) accesses and parses the string values of the arguments, but *USE ARG* allows nonstring arguments and does a one-to-one assignment of arguments to REXX variables. This is the way we pass objects (not only strings) between routines.

New and Enhanced Built-In Functions

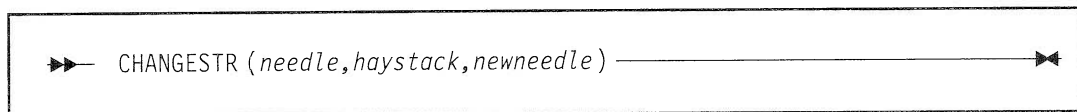
Object REXX has three new built-in functions and some changes to nine old ones.

ARG (Enhanced)



ARG has two new options. The first is *array*, which returns the arguments in the form of an array object. The array index corresponds with the argument position. If the option *n* is used, the index starts at the specified position. If any argument is omitted, the corresponding index is absent. The second new option is *normal*, which returns the *nth* argument, if it exists, or the null string otherwise.

CHANGESTR (New)



CHANGESTR returns a copy of *haystack*, in which *newneedle* replaces all occurrences of *needle*.

CONDITION (Enhanced)

```
➤➤  CONDITION ( option ) ➤➤
```

CONDITION has two new options. The *additional* option makes it possible to get some additional object information on certain conditions (*NOMETHOD*, *NOSTRING*, *NOTREADY*, *SYNTAX*, and *USER*). The second new option, *object*, returns an object containing all the information about the current trapped condition. This can be used to create a generalized trap-and-debug routine, as described in *CALL/SIGNAL (Enhanced)* on page 280.

COUNTSTR (New)

```
➤➤  COUNTSTR (needle, haystack) ➤➤
```

COUNTSTR returns a count of the nonoverlapping occurrences of *needle* in *haystack*. Here is one example:

```
countstr('11','101111101110')  --> 3  /* observe - no overlap */
```

DATATYPE (Enhanced)

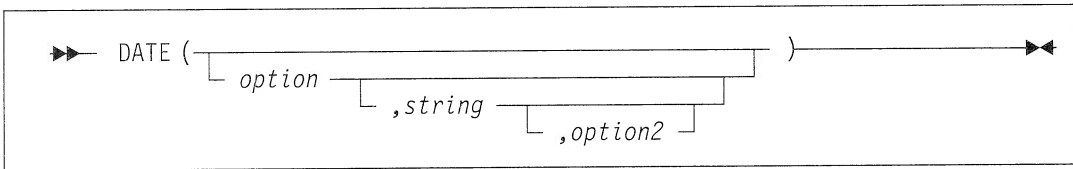
```
➤➤  DATATYPE (string ,type) ➤➤
```

DATATYPE has two new types. The first one is *variable*. As an example, *DATATYPE*(*xyz*,*v*) would return 1 if *xyz* could be on the left-hand side of an assignment without causing a *SYNTAX* condition.

The second new type is *9 Digits*. The description specifies that this type returns 1 if *DATATYPE*(*string*,*w*) would return 1 when *NUMERIC DIGITS* is set to 9. Thus if *NUMERIC DIGITS* is larger than 9, type 9 returns 0 for any whole number larger than 9 digits. Here is an example:

```
numeric digits 12
datatype('1234567890','W')  --> 1  /* less than digits() */
datatype('1234567890','9')  --> 0  /* more than 9 digits */
```

DATE (Enhanced)



`DATE` is now enhanced so that it is possible to work with a date other than the current one. The *string* allows input of a date to translate from one form to another. If the input string is not in the default format (*dd mon yyyy*), *option2* can be used to specify the format to Object REXX. For example, if you want to know how many days it is to your next birthday, enter the following statement in a *REXXTRY* window (*96/mm/dd* is your birthday):

```
say date('B','96/mm/dd','0') - date('B') 'days'
```

Two of the old options have different names. *Basedate* is now only *base* and *sorted* is changed to *standard*.

STREAM (Enhanced)

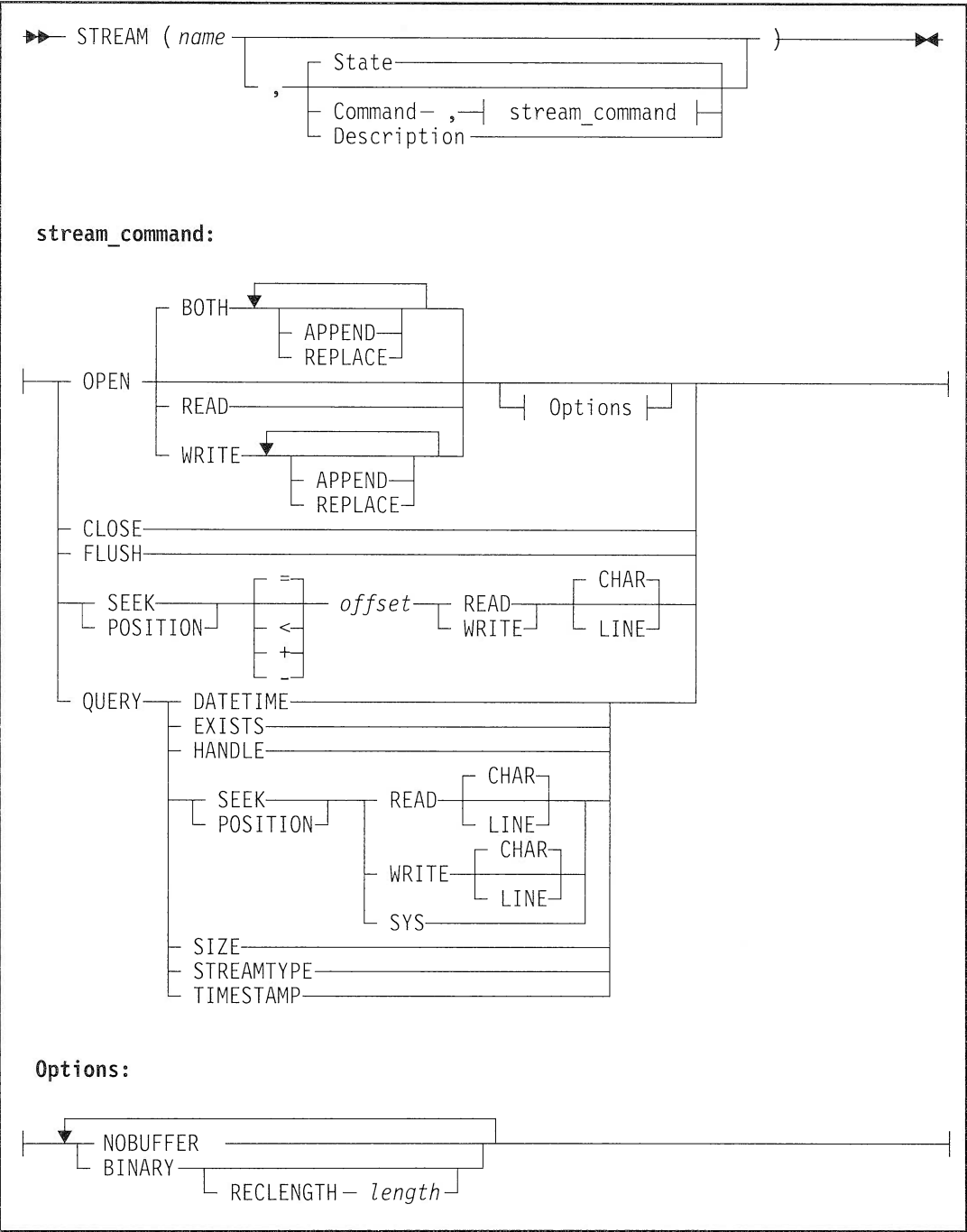
In Object REXX, input and output can be handled two ways. The old way is to use the built-in functions (*STREAM*, *LINES*, *LINEIN*, *LIN- EOUT*, *CHARIN*, and *CHAROUT*), which still work. *STREAM* has a lot of new command strings that we will look at, but we will not go through them all in detail. The new way is to use the new stream class (*.Stream*) in Object REXX, in which all of the built-in functions are available through methods.

Whichever we choose, we must remember not to mix the two ways for the same stream object. When we use the built-in I/O functions, the language processor creates a stream object and maintains it for us. If we use the *new* method to create a stream object, the object is returned to and maintained by our own program.

Because of this, when Object REXX stream methods and stream built-in functions refer to the same file from the same program, there are two separate stream objects with different read and write pointers. This will cause unpredictable results if the stream is written to by using both methods and built-in functions.

So what are the changes to *STREAM* that both methods and functions can use:

1. *OPEN* has some new options. First, Object REXX now supports separate pointers for read and write. The default is to open for both read and write. That can also be specified by option *both*, in case we want to point it out or add one of the new position options, *append* or *replace*. The position options are also valid if we open for *WRITE*.

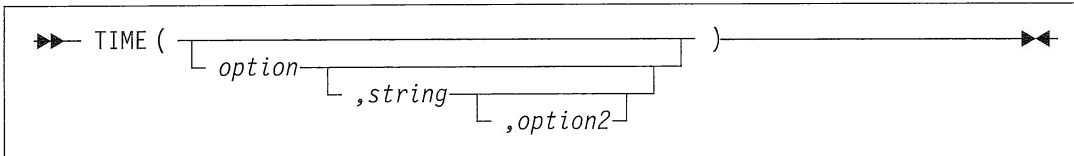


Option *nobuffer* turns off buffering of the stream. This forces all data written to the stream to be physically written immediately to the media.

Binary makes it possible to handle data without regard to any line-end characters, and *reclength* makes it possible to define a fixed record length so that line operations can be used.

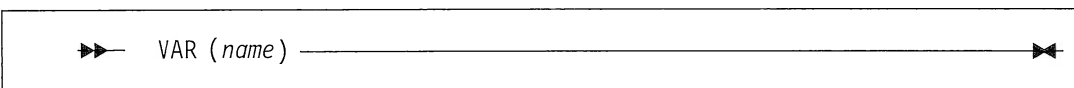
2. *FLUSH* is a new command that forces any data currently buffered for writing to be written to this stream.
3. *SEEK* now has a synonym called *POSITION*. Since we now have two pointers, we have to choose between the *read* pointer (default) and the *write* pointer. *Char* (default) specifies that we are seeking in terms of character position, and *line* in terms of lines.
4. *QUERY* is enhanced by four new options:
 - *Handle*—returns the handle associated with the open stream.
 - *Seek/position*—returns the current read or write position of the file, as qualified by *read*, *write*, *char*, and *line*.
 - *Streamtype*—returns the type of stream (*persistent*, *transient*, or *unknown*).
 - *Timestamp*—returns the date and time stamps of a stream in the form *yyyy-mm-dd hh:mm:ss*.

TIME (Enhanced)



TIME is now enhanced so that it is possible to work with a time other than the current one. The *string* allows input of a date to translate from one form to another. If the input string is not in the default format (*hh:mm:ss*), *option2* can be used to specify the format to Object REXX.

VAR (New)

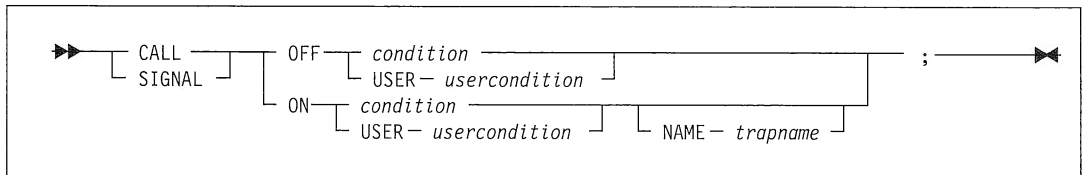


VAR is a new built-in function. It returns 1 if *name* is the name of a variable (that is, a symbol that has been assigned a value), or 0 otherwise.

New Condition Traps

New condition traps are implemented in both the *CALL* and *SIGNAL* instructions.

CALL/SIGNAL (Enhanced)



The new conditions are explained in *SIGNAL (Enhanced)* on page 274. Note that the *RAISE* condition does not trap on the level issued. It shows up as a trap on the calling statement in the parent routine.

The code examples below show the use of a generalized trap routine. A main program requires the class definition and a generalized trap routine. It creates an object and runs a method that causes a syntax error.

The main program:

```

/* TstRaise.Cmd - Test the new RAISE instruction */
signal on any
tm = .myTest~new
say tm~myMethodA
exit
any: interpret .local["M.TRAPDSP"]
::requires 'TstRaise.CaM'      /* myTest class and methods */
::requires 'TrapDisp.Cmd'     /* generalized trap routine */

```

The program containing the Object REXX class and methods:

```

/* TstRaise.CaM - Class & Method directives for TstRaise.Cmd */
::class myTest public
::method init
  return
::method myMethodA
  signal on any
  x = self~myMethodB
  return x
  any: interpret .local["M.TRAPRTN"]
::method myMethodB
  signal on any
  a = 'xyz'
  c = a+2                      /* this line will cause SYNTAX error */
  return c
  any: interpret .local["M.TRAPRTN"]

```


The generalized trap routine:

```
/* TrapDisp.Cmd - Error condition trap and display routines */
.local["M.TRAPRTN"] = 'trace "o"; ',
'if .local["M.SIGL"] = .nil then do; ',
'  .local["M.SIGL"] = sigl; ',
'  .local["M.COBJ"] = condition("O"); ',
'  PARSE SOURCE with . sourceid; ',
'  .local["M.COBJ"]["M.MODULE"] = sourceid; ',
'  .local["M.COBJ"]["M.LINE"] = sourceline(sigl); ',
'end; ',
'raise propagate; '
.local["M.TRAPDSP"] = 'trace "o"; ',
'signal off any; ',
'if .local["M.SIGL"] <> .nil then do; ',
'  sigl = .local["M.SIGL"]; ',
'  .local["M.SIGL"] = .nil; ',
'  CObj = .local["M.COBJ"]; ',
'end; ',
'else do; ',
'  CObj = condition(o); ',
'  CObj["M.MODULE"] = CObj["PROGRAM"]; ',
'  CObj["M.LINE"] = sourceline(sigl); ',
'end; ',
'if var("rc"); ',
'  then say "REXX ["CObj["CONDITION"]] error" rc ',
'    "in line" sigl:" "ERRORTXT"(rc); ',
'  else say "REXX ["CObj["CONDITION"]] error in line" sigl; ',
'say "The Source Module is: "CObj["M.MODULE"]; ',
'say "Source line is: " CObj["M.LINE"]; ',
'exit; '
```

Sample execution:

```
[C:\]TstRaise
REXX [SYNTAX] error 41 in line 16: Bad arithmetic conversion
The Source Module is: C:\TstRaise.CaM
Source line is:    c = a+2    /* this line will cause SYNTAX error */
```

New REXX Utilities

A set of new REXX utilities has been added in Object REXX. These are described in detail in the Object REXX manuals; therefore, we include only a short description here.

Utilities for WPS

SysOpenObject

Opens a view of an existing WPS object and returns the **WinOpenObject** return codes; 1 (true) if the object was opened, or 0 (false) otherwise.

SysCopyObject

Copies an existing WPS object to the specified destination folder and returns the **WinCopyObject** return codes: 1 (true) if the object was copied, or 0 (false) otherwise.

SysMoveObject

Moves an existing WPS object to the specified destination folder and returns the **WinMoveObject** return codes: 1 (true) if the object was moved, or 0 (false) otherwise.

SysSaveObject

Saves an existing WPS object. The WPS operates asynchronously. When one process updates an object, other processes may not see these updates until the WPS updates the object. You can use the **SysSaveObject** function synchronously to ensure that the WPS updates the object before continuing with other processing. **SysSaveObject** returns the **WinSaveObject** return codes: 1 (true) if the object was saved, or 0 (false) otherwise.

SysCreateShadow

Shadows an existing WPS object to the specified destination folder and returns the **WinCreateShadow** return codes: 1 (true) if the object was shadowed, or 0 (false) otherwise.

Utilities or Semaphores

SysCreateEventSem

Creates or opens an OS/2 event semaphore. Returns an event semaphore handle that can be used with the **DosOpenEventSem**, **DosCloseEventSem**, **DosResetEventSem**, **DosPostEventSem**, and **DosWaitEventSem** functions. Returns a null string if the semaphore cannot be created or opened.

SysOpenEventSem

Opens an OS/2 event semaphore and returns the **DosOpenEventSem** return codes.

SysPostEventSem

Posts an OS/2 event semaphore and returns the **DosPostEventSem** return codes.

SysWaitEventSem

Waits on an OS/2 event semaphore and returns the **DosWaitEventSem** return codes.

SysResetEventSem

Resets an OS/2 event semaphore and returns the **DosResetEventSem** return codes.

SysCloseEventSem

Closes an OS/2 event semaphore and returns the **DosCloseEventSem** return codes.

SysCreateMutexSem

Creates or opens an OS/2 mutex semaphore. Returns a mutex semaphore handle that can be used with the **DosOpenMutexSem**, **DosCloseMutexSem**, **DosRequestMutexSem**, and **DosReleaseMutexSem** functions. Returns a null string if the semaphore cannot be created or opened.

SysOpenMutexSem

Opens an OS/2 mutex semaphore and returns the **DosOpenMutexSem** return codes.

SysRequestMutexSem

Requests an OS/2 mutex semaphore and returns the **DosRequestMutexSem** return codes.

SysReleaseMutexSem

Releases an OS/2 mutex semaphore and returns the **DosReleaseMutexSem** return codes.

SysCloseMutexSem

Closes an OS/2 mutex semaphore and returns the **DosCloseMutexSem** return codes.

Utilities for REXX Macros

SysAddRexxMacro

Adds a routine to the REXX macrospace and returns the **RexxAddMacro** return codes.

SysQueryRexxMacro

Queries the existence of a macrospace function. Returns either the placement order of the macrospace function or a null string if the function does not exist in the macrospace.

SysReorderRexxMacro

Changes the search-order position of a loaded macrospace function. The new search-order position could be either before or after any registered functions and external REXX files. **SysReorderRexxMacro** returns the **RexxReorderMacro** return codes.

SysDropRexxMacro

Removes a routine from the REXX macrospace and returns the **RexxDropMacro** return codes.

SysClearRexxMacroSpace

Removes all loaded routines from the REXX macrospace and returns the **RexxClearMacro** return codes.

SysLoadRexxMacroSpace

Loads all functions from a file created with the **SysSaveRexxMacroSpace** utility. If any of the functions already exists in the macrospace, the entire load request is discarded and the macrospace remains unchanged. **SysLoadRexxMacroSpace** returns the **RexxLoadMacro** return codes.

SysSaveRexxMacroSpace

Saves all REXX macrospace functions to a file. Observe that saved macrospace can be loaded only with the same interpreter level that created the image. **SysSaveRexxMacroSpace** returns the **RexxSaveMacro** return codes.

Utilities for Files

SysAddFileHandle

Adds to the number of file handles available to the current process and returns the number of file handles now available.

SysSetFileHandle

Sets the maximum number of file handles available to the current process and returns the **DosSetMaxFH** return codes.

Utilities for Code Pages

SysQueryProcessCodePage

Queries the current code page for the process. Returns the current code page that the process is using.

SysSetProcessCodePage

Changes the current code page for the process. This change does not affect the display or keyboard code page. **SysSetProcessCodePage** returns the **DosSetProcessCp** return codes.

Utilities for OS/2 Systems

SysBootDrive

Returns the drive used to boot OS/2, for example, C:.

SysElapsedTime

Returns a time in the format: *ssssssss.uuuuuu*. The number has no leading zeros or blanks. The fractional part always has six digits. This function uses the OS/2 high-frequency timer services. It has higher timer resolution than does the REXX built-in *TIME()* function.

SysFileSystemType

Returns the name of the file system for a drive (FAT, HPFS, LAN, ...). If the drive is not accessible, a null string is returned.

SysGetCollate

Retrieves the country-specific collating table. Returns the 256-byte collating-sequence table for the indicated country and code-page combination.

SysLoadFuncs

Loads all RexxUtil functions (or other packages). After a REXX program calls **SysLoadFuncs**, the RexxUtil functions are available in all OS/2 sessions:

```
call RxFuncAdd 'SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs'
call SysLoadFuncs
```

SysMapCase

Performs a national language uppercase mapping to a string. Returns the original string, case-mapped according to the country and code-page combination.

SysNationalLanguageCompare

Compares two character strings, using a country-specific collating table. The strings are compared for the length of the shorter string. Returns the comparison result as 0 (equal), 1 (first string longer or collating higher), -1 (first string shorter or collating lower).

SysProcessType

Returns the type of process in which the REXX program is running. The return values are 0 (full screen), 1 (requires real mode), 2 (VIO windowable), 3 (Presentation Manager), 4 (detached process).

SysQueryEAList

Retrieves the complete list of extended attribute names for a file or directory. Returns the names in a stem variable collection where the *stem.0* entry contains the number of names.

SysSetPriority

Changes the priority of the current process and returns the **DosSetPriority** return codes.

SysShutDownSystem

Shuts down the OS/2 system. Returns 1 for a successful shutdown or 0 for an unsuccessful shutdown.

SysWildCard

Produces an OS/2 edited file name using a source file name and a wildcard editing pattern. Returns the result of editing the source with the wildcard. The editing is performed using the **DosEditName** function.

Migration Considerations

Migration considerations are described in detail in *Object REXX Reference for OS/2*. Here we provide a short extract:

Stems Stems behave a little differently in Object REXX. The symbol functions return *VAR* (not *LIT*) because a stem object is automatically created the first time used, and a *NOVALUE* condition is never raised. Stems can be assigned to each other (*a. = z.*), and they point to the same object.

In many cases, it may be desirable to use some of the new collections provided by Object REXX, instead of a stem variable.

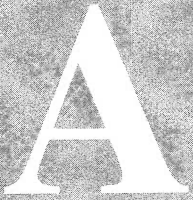
Parse version

Return *n.nn*, the current version.

Streams Avoid mixing methods (*aStream~linein*) and functions (*linein(aStream)*) because they work on different objects representing the same file. *LINEIN*, *CHARIN*, *LINES*, and *CHARS* return the null string for a nonexistent file, but they also create an empty file on the disk.

Earlier error detection

Before the program is started, Object REXX performs some syntax checking and the program might never get control. For example, missing *END* statements and missing parameters are detected before starting the program.



Car Dealer Source Code

This appendix describes the location of all the source code of the sample applications and contains an extract of the source code of the car dealer application.

The following sections of the car dealer source code are listed in this appendix:

- ☐ Sample data
- ☐ Classes and methods
- ☐ Running the car dealer programs

Note: Only an extract of the programs is listed here. All the programs are available on the CD, and on your hard drive once the sample applications have been installed.

Directory Structure

The sample applications are stored in two main directories, one for the car dealer application, and one for the philosopher's forks application.

Car Dealer Application

The car dealer application is available in the CARDEAL directory on the CD and in a directory of your choice when installed on your hard drive. The subdirectories and a description of their content are listed in Table 37.

Table 37. Subdirectories of the Car Dealer Application		
Subdirectory	Description	Details
(main)	Master directory of car dealer application	Table 19 on page 254
SampData	Master files with sample data	Table 20 on page 255
Base	Base class definitions for objects in storage	Table 21 on page 255
FAT	Class definitions for persistent objects in files	Table 22 on page 255
DB2	Class definitions for persistence in DB2	Table 23 on page 256
RAM	Class definitions for objects in RAM	Table 24 on page 256
AUI	Class definitions for ASCII interface	Table 25 on page 256
Media	Multimedia files	Table 26 on page 257
DrDialCD	GUI definitions and executable for Dr. Dialog	Table 27 on page 257
VisProCD	GUI definitions and executable for VisPro/REXX	Table 28 on page 257
VxRexxCD	GUI definitions and executable for Watcom VX.REXX	Table 29 on page 257
SOM	Implementation of Part class in SOM	Table 30 on page 258
StorProc	Sample commands for stored procedures in a DB2 client/server environment	Table 31 on page 258
WPS	Sample commands to visualize car dealer using OS/2 WPS	Table 32 on page 258
WWW	Car dealer on the World Wide Web	Table 33 on page 259
Xamples	Additional small examples	Table 34 on page 259
Install	Installation programs and DB2 setup	Table 35 on page 259

Philosopher's Forks

The philosopher's fork application is available in the PHILFORK directory on the CD and in a directory of your choice when installed on your hard drive. The subdirectories and a description of their content are listed in Table 38.

Table 38. Subdirectories of the Philosopher's Forks Application	
Filename	Description
(main)	Master directory of philosopher's forks (see Table 36 on page 260)
DrDialPF	Subdirectory for Dr. Dialog application
VisProPF	Subdirectory for VisPro/REXX application
VxRexxPF	Subdirectory for Watcom VX-REXX application
ZdialFun	Subdirectory for funny-faces application

Car Dealer Source Code

Sample Data

Note: The not signs (¬) represent tab characters in the sample data listings below.

Sample Customer Data

```

/*-----*/
/* SampData\customer.dat  CarDealer - Customer data file  ITS0-SJC */
/*-----*/
/*number name          address                                     */
/*-----*/
101¬Senator, Dale      ¬Washington
102¬Akropolis, Ida    ¬Athens
103¬Dolcevitā, Felicia ¬Rome
104¬DuPont, Jean      ¬Paris
105¬Deutsch, Hans     ¬Stuttgart
106¬Helvetia, Toni     ¬Zurich
107¬Rising Star       ¬Hollywood
108¬Zabrowski, Russkie ¬Moscow
109¬Valencia, Maria de ¬Barcelona
601¬Wahli, Ueli       ¬ITS0 San Jose
602¬Turton, Trevor    ¬Johannesburg
603¬Grihorn, Eddie    ¬Stockholm
604¬Furukawa, Norio   ¬Tokyo
/*-----*/
999¬New and used cars ¬For sale

```

Figure 111. Sample Customer Data (*sampdata\customer.dat*)

Sample Vehicle Data

```

/*-----*/
/* SampData\vehicle.dat CarDealer - Vehicle data file      ITS0-SJC */
/*-----*/
/*serial make      model      year customer      */
/*-----*/
123456~Ford      ~T      ~1931~101
297465~Volkswagen ~Camper ~1971~102
111111~Porsche   ~Targa  ~1989~102
222222~Lamborghini~Countach ~1992~103
398674~Cadillac  ~Allante ~1991~103
334455~Chevrolet ~Impala  ~1985~104
456456~Toyota    ~Camry   ~1988~105
543543~Pontiac   ~Firebird ~1979~106
911911~Chrysler  ~Le Baron ~1982~106
298653~Mercury    ~Sable   ~1987~106
176549~Oldsmobile ~Aurora  ~1993~107
199999~Acura      ~Legend  ~1990~107
777777~Mercedes   ~380S    ~1990~108
666888~Lincoln    ~Towncar ~1986~109
601001~Audi        ~5000-Wagon~1984~601
602002~BMW         ~735S    ~1991~602
603003~Saab        ~9000    ~1992~603
604004~Nissan      ~Altima  ~1994~604
/*-----new/used cars-----*/
999001~Ford      ~Windstar ~1995~999
999002~Audi       ~V8 Quattro~1990~999
999003~Volvo      ~860 Wagon ~1995~999
999004~Honda      ~Civic    ~1994~999
999005~MixedStuff ~Fun      ~1995~999
/*-----not a car!-----*/
999666~ThinkPad   ~701      ~1995~999
999999~0RexxRedbook~Team     ~1995~999

```

Figure 112. *Sample Vehicle Data (sampdata\vehicle.dat)***Sample Work Order Data**

```

/*-----*/
/* SampData\workord.dat CarDealer - WorkOrder data file    ITS0-SJC */
/*-----*/
/*number date cost complete custmr serial serv.items      */
/*-----*/
1~09/06/95~-1~0~101~123456~1
2~09/07/95~-1~0~103~398674~10~9~4
3~09/08/95~-1~0~106~911911~7~6
4~09/09/95~-1~0~108~777777~11
5~08/01/95~100~1~107~199999~2~3

```

Figure 113. *Sample Work Order Data (sampdata\workord.dat)*

Sample Service Item Data

```

/*-----*/
/* SampData\service.dat  CarDealer - ServiceItem data file  ITS0-SJC */
/*-----*/
/*number description      labor part quant part quant      */
/*-----*/
1 -Brake job              -110-21-1-22-2-23-2-24-2
2 -Check fluids           -25 -10-5-11-1-31-1
3 -Tire rotate/balance    -20
4 -Tires new Sedan        -0 -51-4
5 -Tires new Sport        -10 -52-4
6 -Starter                -75 -71-1
7 -Alternator             -90 -72-1
8 -Heating system         -145-61-1-62-1-81-1-82-1
9 -Electrical             -85 -45-3-91-1
10-Exhaust system         -85 -1-1
11-Fenders                -45 -41-2

```

Figure 114. Sample Service Item Data (*sampdata\service.dat*)

Sample Part Data

```

/*-----*/
/* SampData\part.dat      CarDealer - Part data file      ITS0-SJC */
/*-----*/
/*number description cost stock      */
/*-----*/
1-Muffler                 -120-3
10-Oil 10-40 quart        -5-30
11-Oil filter             -22-15
21-Brake cylinder         -120-3
22-Brake fluid            -7-13
23-Brake drum             -28-6
24-Brake disk             -35-9
31-Steering fluid         -8-40
41-Fender                 -67-2
45-Light bulb             -2-20
51-Tire 185-70            -57-8
52-Tire 205-60            -73-12
61-Belt                   -12-2
62-Radiator               -133-1
71-Starter                -189-4
72-Alternator             -165-2
81-Water pump             -97-1
82-Heating control        -43-1
91-Cruise control         -54-2

```

Figure 115. Sample Part Data (*sampdata\part.dat*)

Multimedia Setup

Multimedia Data Definition File

```

/*-----*/
/* Media\media.dat      CarDealer - Multimedia definition      ITS0-SJC */
/*-----*/
/* serial, title of file      , filename                        */
/*-----*/
999001, Fact-sheet      , ford.fac
999001, Side picture    , fordsid.bmp
999001, Front picture   , fordfrt.bmp
999001, Back picture    , fordback.bmp
999001, Angle picture   , fordang.bmp
999001, Audio           , ford.wav
999002, Fact-sheet      , audi.fac
999002, Side picture    , audisid.bmp
999002, Front picture   , audifrt.bmp
999002, Back picture    , audiback.bmp
999002, Audio           , audi.wav
999003, Fact-sheet      , volvo.fac
999003, Side picture    , volvosid.bmp
999003, Front picture   , volvofrt.bmp
999003, Back picture    , volvoback.bmp
999003, Audio           , volvo.wav
999004, Fact-sheet      , honda.fac
999004, Side picture    , hondasid.bmp
999004, Front picture   , hondafrt.bmp
999004, Back picture    , hondaback.bmp
999004, Audio           , honda.wav
999005, Fact-sheet      , mixed.fac
999005, Tow truck       , towtruck.bmp
999005, Truck           , truck.bmp
999005, Pickup          , pickup.bmp
999005, Fire engine     , fireeng.bmp
999005, Motor cycle     , motocycl.bmp
999005, Audio           , mixed.wav
999666, Fact-sheet      , ibm701i.fac
999666, ThinkPad 701    , ibm701i.bmp
999666, Video           , ibm701i.avi
999999, Fact-sheet      , orexxred.fac
999999, Team Photo      , orexteam.bmp
999999, Ueli Wahli      , ueli.bmp
999999, Trevor Turton   , trevor.bmp
999999, Eddie Griborn   , eddie.bmp
999999, Norio Furukawa  , norio.bmp
999999, Audio           , orexxred.wav
999999, Video           , macaw.avi
601001, Fact-sheet      , wahlii.fac
601001, Ueli's Portrait , ueli2.bmp
601001, Ueli's car      , audi.bmp
601001, License plates  , licenses.bmp
601001, Cactus garden   , cactus.bmp
601001, Family cat      , boxie.bmp
601001, Cat in trouble  , cat.bmp
601001, Audio           , wahlii.wav
602002, Trevor's Portrait, trevor2.bmp
603003, Eddie's Portrait, eddie2.bmp
604004, Norio's Portrait, norio2.bmp
end

```

Figure 116. *Multimedia Data Definition File (media\media.dat)*

Base Classes

Base Customer Class

```

/*-----*/
/* Base\carcust.cls  CarDealer - Customer class (base)    ITS0-SJC */
/*-----*/

::class CustomerBase public

    /*----- class methods -----*/

::method initialize class                                /* prepare class          */
    expose extent
    extent = .set~new                                     /* - keep track of cust.  */
    self~persistentLoad                                  /* - and load into memory */

::method add class                                       /* add new customer        */
    expose extent
    use arg custx
    if custx~class = self then do                        /* - check if already there */
        do custo over extent
            if custo~number = custx~number then return
        end
        extent~put(custx)                                /* - add to extent          */
    end

::method remove class                                   /* remove customer from     */
    expose extent                                       /*      extent              */
    use arg custx
    if custx~class = self then
        extent~remove(custx)

::method findNumber class                               /* find customer by number */
    expose extent
    parse arg custnum
    do custx over extent                                /* - search extent          */
        if custx~number = custnum then return custx      /* - return when found      */
    end
    return .nil

::method findName class                                 /* find customer by name   */
    arg custsearch
    custnames = .list~new                                /* - prepare result list    */
    do custx over self~extent                            /* - check extent           */
        if abbrev(translate(custx~name),custsearch) then do
            custstring = custx~number~right(3)|| ,
                '- 'custx~name'- 'custx~address
            custnames~insert(custstring)                  /* - add a match            */
        end
    end
    return custnames~makearray                            /* - return result array    */

```

Figure 117. (Part 1 of 4) *Base Customer Class (base\carcust.cls)*

```

::method findAddress class                                /* find customer by address*/
  arg custsearch
  do custx over self~extent                               /* - check extent          */
    if custx~address = custsearch then
      return custx~number                                /* - return customer number*/
    end
  return ''                                                /* - return not found      */

::method extent class                                    /* return extent of cust.  */
  expose extent
  return extent~makearray                                /* - as an array           */

::method heading class                                   /* return a heading        */
  return 'Number Name Address'

/*----- instance methods -----*/

::method init                                             /* initialize new customer */
  expose customerNumber name address cars orders
  self~init:super                                         /* - call parent           */
  use arg customerNumber, name, address
  cars = .set~new                                         /* - prepare cars/orders   */
  orders = .set~new
  if arg() < 3 | arg() > 4 then return self~setnil
  if \datatype(customerNumber,'W') then return self~setnil
  if customerNumber<100 | customerNumber>999 then return self~setnil
  self~class~add(self)                                    /* - add to extent         */
  if arg() = 4 then self~persistentInsert                /* - a real new customer   */

::method setnil private                                  /* set customer data nil   */
  expose customerNumber name address cars orders
  self~class~remove(self)                                /* - remove from extent    */
  cars = .nil
  orders = .nil
  customerNumber = 0
  name = '-none-'
  address = '-none-'
  return .nil

::method delete                                           /* delete a customer       */
  expose cars orders
  do carx over cars                                       /* - delete all cars       */
    carx~delete
  end
  do workx over orders                                    /* - delete all workorders */
    workx~delete
  end
  self~class~remove(self)                                /* - remove from extent    */
  self~persistentDelete                                  /* - delete permanent stor */
  self~setnil

::method number unguarded                                /*                          */
  expose customerNumber
  return customerNumber

::method name attribute                                  /* customer's name         */

::method address attribute                                /* customer's address      */

::method update                                           /* update customer data    */
  expose name address
  if arg() = 2 then do
    use arg name, address

```

Figure 117. (Part 2 of 4) Base Customer Class (base\carcust.cls)

```

        self~persistentUpdate          /* - update persistent stor */
    end

::method addVehicle                    /* add a vehicle          */
    expose cars
    use arg newcar
    owner = newcar~getowner            /* - check its owner      */
    if owner = self | owner = .nil then do
        cars~put(newcar)              /* - add if no owner      */
        if owner = .nil then
            newcar~setowner(self)      /* - set new owner        */
        end
    else do
        say 'Cannot add car' newcar~makemodel 'to customer' self~name /* - error if other owner */
        say ' it belongs to' newcar~getowner~name
    end

::method removeVehicle                /* remove vehicle from cust */
    expose cars
    use arg oldcar
    oldcar~deleteOwner                 /* - delete owner         */
    cars~remove(oldcar)                /* - remove from cars     */

::method checkVehicle                 /* check if car in set     */
    expose cars
    use arg somecar
    if cars~hasindex(somecar) then return 1 /* - yes it is          */
    else return 0

::method getVehicles                  /* return array of cars    */
    expose cars
    return cars~makearray

::method findVehicle                  /* find car by serial      */
    expose cars
    use arg serial
    do carx over cars                  /* - check all cars       */
        if carx~serial = serial then return carx
    end
    return .nil

::method addOrder                     /* add order to customer   */
    expose orders
    use arg newwork
    orders~put(newwork)                /* - add order to set     */

::method removeOrder                  /* remove order from cust. */
    expose orders
    use arg oldwork
    orders~remove(oldwork)             /* - remove order from set */

::method getOrders                    /* return all orders       */
    expose orders
    return orders~makearray            /* - as an array          */

::method detail                       /* return a detail line    */
    expose customerNumber name address
    return customerNumber~right(5) ' ' name~left(20) ' ' address~left(20)

::method makestring                   /* default string output   */
    expose customerNumber name
    return 'Customer:' customerNumber name

::method display                      /* display customer data   */

```

Figure 117. (Part 3 of 4) *Base Customer Class (base\carcust.cls)*

```

expose customerNumber name address cars orders
say '- '~copies(78)
say self~class~heading
say self~detail
if cars~items > 0 then
  do carx over cars
    say ' Vehicle:' carx~detail
  end
if orders~items > 0 then do
  do orderx over orders
    say ' WorkOrder:' orderx~detail
  end
end
end

```

Figure 117. (Part 4 of 4) *Base Customer Class (base\carcust.cls)*

Base Vehicle Class

```

/*-----*/
/* Base\carvehi.cls CarDealer - Vehicle class (base) ITS0-SJC */
/*-----*/

::class VehicleBase public

/*----- class methods -----*/

::method initialize class /* prepare class */
self~persistentLoad /* - load into memory */

/*----- instance methods -----*/

::method init /* initialize new vehicle */
expose serialNumber make model year owner
self~init:super
use arg serialNumber, make, model, year, owner
if arg() < 5 | arg() > 6 then self~setnil
if owner \= .nil then
  owner~addVehicle(self) /* - add car to customer */
if arg() = 6 then self~persistentInsert /* - insert real new car */

::method setnil private /* set vehicle data nil */
expose serialNumber make model year owner
if owner \= .nil then
  owner~removeVehicle(self) /* - remove from customer */
serialNumber = 0
make = '-none-'
model = '-none-'
year = 0
owner = .nil

::method delete /* delete a vehicle */
expose serialNumber make model year owner
self~persistentDelete /* - from permanent stor */
self~setnil

::method serial /* return serial number */
expose serialNumber
return serialNumber

```

Figure 118. (Part 1 of 2) *Base Vehicle Class (base\carvehi.cls)*


```

::method make attribute          /* vehicle's make      */
::method model attribute         /* vehicle's model   */
::method year attribute          /* vehicle's year    */
::method update                  /* update vehicle data */
  expose make model year
  if arg() = 3 then do
    use arg make, model, year
    self~persistentUpdate
  end
  /* - in permanent storage */

::method makemodel unguarded     /* return make and model */
  expose make model
  return make~strip~'-'~model~strip
  /* - as string */

::method getOwner unguarded      /* return owner (customer) */
  expose owner
  return owner

::method setOwner                /* set a new owner (cust) */
  expose owner
  use arg newowner
  if owner = .nil then
    if newowner~checkVehicle(self) then
      use arg owner
  /* - if its the proper one */

::method deleteOwner             /* delete the owner (cust) */
  expose owner
  owner = .nil

::method detail                  /* return a detail line */
  expose serialNumber make model year
  return serialNumber~right(8) ' ' make~left(12) ' ' model~left(10) ,
        ' ' year

::method makestring              /* default string output */
  expose serialNumber make model
  return 'Vehicle:' serialNumber make model

::method display                 /* display vehicle data */
  expose serialNumber make model year owner
  if owner = .nil then ownerst = '-no owner-'
  else ownerst = owner~number
  say serialNumber~right(8) ' ' make~left(12) ' ' model~left(10) ,
    ' ' year ' ' ownerst

```

Figure 118. (Part 2 of 2) *Base Vehicle Class (base\carvehi.cls)*

Base Work Order Class

```

/*-----*/
/* Base\carwork.cls      CarDealer - WorkOrder class (base) ITS0-SJC */
/*-----*/

```

Figure 119. (Part 1 of 6) *Base Work Order Class (base\carwork.cls)*

```

::class WorkOrderBase public
/*-----class methods-----*/
::method initialize class /* prepare the class */
  expose extent WorkServRel
  extent = .list~new /* - extent of work orders */
  if .local['Cardeal.WorkServRel'] = .nil then /* - prepare relation to */
    .local['Cardeal.WorkServRel'] = .Relation~new
  WorkServRel = .local['Cardeal.WorkServRel'] /* - service items */
  self~persistentLoad /* - load into memory */
::method getWorkServRel class /* return the relation */
  expose WorkServRel
  return WorkServRel

::method add class /* add workorder to extent */
  expose extent
  use arg workx
  if workx~class = self then do
    do worko over extent /* - check if already there*/
      if worko~number = workx~number then return worko~getIndex
    end
    return extent~insert(workx, .nil) /* - insert new at start */
  end

::method remove class /* remove order from extent*/
  expose extent
  use arg indx, workx
  if extent~at(indx) = workx then /* - ccheck and remove */
    extent~remove(indx)

::method findNumber class /* find workorder by number */
  expose extent
  use arg worknum
  do workx over extent /* - check the extent */
    if workx~number = worknum then return workx
  end
  return .nil

::method findStatus class /* find workorder by status */
  expose extent
  use arg xstatus
  worklist = .list~new /* - prepare result */
  xstat1 = 0 /* - 0 is incomplete */
  xstat2 = 1 /* - 1 is complete */
  if xstatus = 0 then xstat2=0
  if xstatus = 1 then xstat1=1
  do workx over extent /* - go over all orders */
    xstatus = workx~getStatus /* - and check the status */
    if xstatus >= xstat1 1059 xstatus <= xstat2 then
      if xstatus = 0 then statusx = 'Incomplete'
      else statusx = 'Complete'
      workstring = workx~number~left(3) ' ' workx~date ,
        workx~cost~right(6) statusx~left(11) ,
        (workx~getvehicle~make~strip || ,
        '- 'workx~getvehicle~model~strip)~left(20) ,
        workx~getcustomer~name
      worklist~insert(workstring, .nil) /* - add to result */
    end
  end
  return worklist~makearray /* - return result as array */

```

Figure 119. (Part 2 of 6) Base Work Order Class (base\carwork.cls)

```

::method newNumber class                                /* return a new number */
  expose extent
  if extent~items = 0 then return 1
  newnum = 0
  do workx over extent                                  /* - find maximum number */
    newnum = max(newnum, workx~number)
  end
  return newnum + 1                                     /* - return next higher */

::method extent class                                  /* return extent as array */
  expose extent
  return extent~makearray

/*-----instance methods-----*/
::method init                                           /* initialize new workorder*/
  expose orderNumber cost date status customer car listindex
  self~init:super
  status = 0                                           /* - incomplete */
  cost = -1                                           /* - unknown cost */
  orderNumber = 0
  if arg() = 3 then do                                  /* - new work order */
    use arg date, customer, car
    orderNumber = self~class~newNumber /* - find new number */
    listindex = self~class~add(self) /* - add to extent */
    customer~addOrder(self) /* - add to customer */
    self~persistentInsert /* - add to persistent stor*/
  end
  else if arg() = 6 then do /* - load from persistent */
    use arg orderNumber, date, cost, status, customer, car
    listindex = self~class~add(self) /* - add to extent */
    customer~addOrder(self) /* - add to customer */
  end
  else self~setnil

::method setnil private                                /* set workorder data nil */
  expose orderNumber cost date status customer car listindex
  customer~removeOrder(self)
  self~class~remove(listindex,self) /* - remove from extent */
  status = 0
  cost = -1
  orderNumber = 0
  date = '00/00/00'
  customer = .nil
  car = .nil
  listindex = 0
  return .nil

::method delete                                         /* delete a work order */
  expose orderNumber cost date status customer car listindex
  self~class~remove(listindex,self) /* - remove from extent */
  self~persistentDelete /* - delete persistent stor*/
  self~setnil

::method number unguarded                              /* return workorder number */
  expose orderNumber
  return orderNumber

::method cost unguarded                                /* return cost of workorder*/
  expose cost
  return cost

::method date unguarded                                /* return date of workorder*/
  expose date
  return date

```

Figure 119. (Part 3 of 6) *Base Work Order Class (base\carwork.cls)*

```

::method setstatus                                /* change the status */
  expose status
  use arg newstatus
  if newstatus = 0 | newstatus = 1 then do /* - change peristent stor */
    if status \= newstatus then self~persistentUpdate
    status = newstatus
  end

::method getstatus unguarded                      /* return the status */
  expose status
  return status

::method getstatust unguarded                    /* return status as text */
  expose status
  if status = 0 then return 'incomplete'
  else return 'complete'

::method getindex unguarded private             /* return index in extent */
  expose listindex
  return listindex

::method getCustomer unguarded                  /* return the customer */
  expose customer
  return customer

::method getVehicle unguarded                   /* return the vehicle */
  expose car
  return car

::method getServices                            /* return all services */
  return self~class~getWorkServRel~allat(self)

::method addServiceItem                        /* add service to workorder*/
  use arg itemx
  workserv = self~class~getWorkServRel          /* - get the relation */
  if workserv~hasitem(itemx,self) then return /* - cannot add same item */
  workserv[self] = itemx                         /* - record in relation */
  if arg() = 2 then return self~persistentInsertServ(itemx~number)

::method removeServiceItem                    /* remove a service */
  use arg itemx
  workserv = self~class~getWorkServRel
  workserv~removeitem(itemx,self)               /* - remove in relation */
  if arg() = 2 then return self~persistentDeleteServ(itemx~number)

::method getTotalCost                         /* compute total cost */
  expose cost
  totalcost = 0
  do servx over self~getServices                 /* - sum up all services */
    totalcost = totalcost + servx~laborcost + servx~getPartsCost
  end
  if cost \= totalcost then do                   /* - update cost attribute */
    cost = totalcost
    self~persistentUpdate
  end
  return totalcost

::method checkAndDecreaseStock                /* check if enough parts */
  expose status
  if status = 1 then return 0                   /* - not for complete ones */
  enough = 1
  do servx over self~getServices                 /* - check all services */
    partsx = servx~getparts
    do partx over partsx                         /* - and parts in service */

```

Figure 119. (Part 4 of 6) Base Work Order Class (base\carwork.cls)

```

        quan = servx~getquantity(partx)
        partno = partx~number                /* - get part number */
        if symbol("stock."partno) = 'LIT' then /* - record stock */
            stock.partno = partx~stock
            stock.partno = stock.partno - quan
            if stock.partno < 0 then do          /* - check temporary stock */
                enough = 0
                say 'servx
                say ' --> Not enough stock for' partx
            end
        end
    end
end
if enough then do                            /* - all stocks are OK */
    do servx over self~getServices            /* - go over all services */
        partsx = servx~getparts
        do partx over partsx                  /* - and all parts */
            quan = servx~getquantity(partx)
            x = partx~decreaseStock(quan)      /* - decrease stock of part*/
        end
    end
    status = 1
    x = self~getTotalCost                    /* - and compute total cost*/
end
return enough

::method generateBill                        /* prepare the bill */
    expose orderNumber date customer car
    separ = '-'~copies(78)
    bill = .list~new                          /* - result lines */
    bill~insert('Bill for work order' orderNumber left(' ',30) 'Date:' date)
    bill~insert(separ)
    bill~insert(' Customer:' customer~name)
    bill~insert(' Vehicle:' car~makemodel)
    bill~insert(separ)
    bill~insert('DescriptionParts          Unit      ',
                'Partcost Laborcost')
    bill~insert(separ)
    do servx over self~getServices            /* - over all services */
        bill~insert(servx~description~left(54) servx~getPartsCost~right(8) ,
                    servx~laborcost~right(10))
        partsx = servx~getparts
        do partx over partsx                  /* - and parts in service */
            quan = servx~getquantity(partx)
            costx = quan * partx~price
            bill~insert(' ' ~left(18) quan~right(3) partx~description~left(16) ,
                        '$' partx~price~right(4) '=' costx~right(5))
        end
    end
    bill~insert(separ)
    bill~insert('Total cost of work order' ~left(65) self~getTotalCost~right(8))
    bill~insert(separ)
    return bill~makearray

::method detail                             /* return a detail line */
    expose orderNumber cost date status
    return orderNumber~right(3) ' Date:' date~left(8) ' Cost:' ,
            cost~right(5) ' Status:' self~getstatust

::method detailcust                          /* return cust/vehicle */
    expose customer car
    return 'Customer:' customer~name~left(20) ,
            'Vehicle:' car~makemodel~left(20)

```

Figure 119. (Part 5 of 6) Base Work Order Class (base\carwork.cls)

```

::method makestring          /* return default string */
  expose orderNumber cost date status customer car
  return 'Workorder:' orderNumber date self~getstatust ,
        ('customer~name~left(10) '/'car~makemodel~left(10)')

::method makeline            /* return a short line */
  expose orderNumber cost date status customer car
  return orderNumber~left(3) ' ' date cost~right(6) ,
        self~getstatust~left(11) car~makemodel customer~name

::method display             /* display work order data */
  expose orderNumber cost date status customer car
  separ = '~'~copies(78)
  say workx~detail
  say workx~detailcust
  first = 1
  do servx over self~getServices
    if first then say ' Services:' servx~number~right(3) servx~description
    else          say '          ' servx~number~right(3) servx~description
    first = 0
    lines = lines + 1
  end

```

Figure 119. (Part 6 of 6) *Base Work Order Class (base\carwork.cls)*

Base Service Item Class

```

/*-----*/
/* Base\carserv.cls      CarDealer - ServiceItem class(base) ITS0-SJC */
/*-----*/

::class ServiceItemBase public

/*----- class methods -----*/

::method initialize class          /* prepare the class */
  expose extent WorkServRel
  extent = .list~new               /* - extent as a list */
  if .local['Cardeal.WorkServRel'] = .nil then /* - prepare relation */
    .local['Cardeal.WorkServRel'] = .Relation~new
    WorkServRel = .local['Cardeal.WorkServRel'] /* - to work orders */
    self~persistentLoad                /* - load into memory */

::method getWorkServRel class      /* return the relation */
  expose WorkServRel
  return WorkServRel

::method add class                 /* add service to extent */
  expose extent
  use arg servx
  if servx~class = self then /* - add to extent */
    return extent~insert(servx)

::method remove class              /* remove service from ext.*/
  expose extent
  use arg indx, servx
  if extent~at(indx) = servx then /* - remove ffrom extent */
    extent~remove(indx)

```

Figure 120. (Part 1 of 3) *Base Service Item Class (base\carserv.cls)*

```

::method findNumber class          /* find service by number */
  expose extent
  parse arg servnum
  if extent~items > 0 then          /* - check the extent */
    do servx over extent
      if servx~number = servnum then return servx
  end
  return .nil

::method extent class              /* return extent as array */
  expose extent
  return extent~makearray

::method heading class             /* return a heading line */
  return 'Item LaborCost Description Quantity Part'

/*----- instance methods -----*/

::method init                      /* initialize new service */
  expose itemNumber description laborCost parts quantity. listindex
  self~init:super
  use arg itemNumber, description, laborCost
  parts = .set~new                  /* - set of parts */
  quantity. = ''                   /* - with quantity */
  if arg() \= 3 then self~setnil
  else listindex = self~class~add(self) /* - add to extent list */

::method setnil private            /* set service data nil */
  expose itemNumber description laborCost parts listindex
  self~class~remove(listindex,self) /* - remove from extent */
  itemNumber = 0
  description = '-none-'
  laborCost = 0
  parts = .nil
  quantity. =
  listindex = 0
  return .nil

::method delete                    /* delete a service item */
  expose listindex
  self~class~remove(listindex,self) /* - remove from extent */
  /* self~persistentDelete */
  self~setnil

::method number unguarded          /* return service number */
  expose itemNumber
  return itemNumber

::method laborcost unguarded       /* return labor cost */
  expose laborCost
  return laborCost

::method description unguarded     /* return description */
  expose description
  return description

::method usesPart                  /* record used part */
  expose parts quantity.
  if arg() \= 2 then return
  use arg partx, quan
  parts~put(partx)                  /* - add to parts list */
  quantity.partx = quan             /* - with quantity */

```

Figure 120. (Part 2 of 3) *Base Service Item Class (base\carserv.cls)*

```

::method getParts                                /* return all parts      */
  expose parts
  return parts~makearray                          /* - as an array      */

::method getQuantity                             /* return quantity     */
  expose quantity.
  use arg partx
  return quantity.partx                          /* - of a part        */

::method getPartsCost                            /* calculate cost of parts */
  expose parts quantity.
  partcost = 0
  do partx over parts                            /* - over all parts    */
    partcost = partcost + partx~price * quantity.partx
  end
  return partcost

::method getWorkOrders                          /* return workorders    */
  return self~class~getWorkServRel~allindex(self)/* using this service */

::method detail                                 /* return detail line    */
  expose itemNumber description laborCost
  return itemNumber~right(3) laborCost~right(11) ' ' description~left(20)

::method makestring                             /* return default string */
  expose itemNumber description laborCost
  return 'ServiceItem:' itemNumber '($'laborCost')' description

::method display                                /* display service data  */
  expose itemNumber description laborCost parts quantity.
  say '-'~copies(78)
  say self~class~heading
  say self~detail
  do partx over parts
    say ' ' ~left(30) quantity.partx~right(6) ' ' ,
      partx~number~right(3) partx~description
  end
  do workx over self~getWorkOrders
    say '-' workx
  end
end

```

Figure 120. (Part 3 of 3) *Base Service Item Class (base\carserv.cls)*

Base Part Class

```

/*-----*/
/* Base\part.ori CarDealer - Part class (base)      ITS0-SJC */
/* (original class, becomes carpart.cls)            */
/*-----*/
.local['Cardeal.Part.som'] = 'No' /* mark as NOT in SOM */

::class PartBase public

/*-----class methods-----*/

```

Figure 121. (Part 1 of 3) *Base Part Class (base\part.ori)*


```

::method initialize class          /* prepare the class      */
  expose extent
  extent = .set~new                /* - extent of parts    */
  self~persistentLoad              /* - load into memory   */

::method add class                 /* add new part to extent */
  expose extent
  use arg partx
  if partx~class = self then      /* - add to extent      */
    extent~put(partx)

::method remove class             /* remove part from extent */
  expose extent
  use arg partx
  if partx~class = self then      /* - remove             */
    extent~remove(partx)

::method findNumber class         /* find part by number    */
  expose extent
  parse arg partnum
  do partx over extent            /* - check the extent    */
    if partx~number = partnum then return partx
  end
  return .nil

::method extent class             /* return extent as array */
  expose extent
  return extent~makearray

::method heading class           /* return a heading line  */
  return 'Partid Description Price Stock'

/*----- instance methods -----*/

::method init                    /* initialize a new part  */
  expose partid description price stock
  self~init:super
  use arg partid, description, price, stock
  if arg() \= 4 1201 arg() \= 5 then self~set
  else self~class~add(self)      /* - add to extent      */
  if arg() = 5 then self~persistentInsert /* -add to persistent */

::method setnil private          /* set part data nil     */
  expose partid description price stock
  self~class~remove(self)        /* - remove from extent  */
  partid = 0
  description = '-none-'
  price = 0
  stock = 0
  return .nil

::method delete                  /* delete a part         */
  self~class~remove(self)        /* - remove from extent  */
  /* self~persistentDelete */    /* - not implemented    */
  self~setnil

::method number unguarded        /* return parts number    */
  expose partid
  return partid

::method price unguarded         /* return price of part   */
  expose price
  return price

```

Figure 121. (Part 2 of 3) Base Part Class (*base\part.ori*)

```

::method description unguarded          /* return description */
  expose description
  return description

::method stock unguarded                /* return stock of part */
  expose stock
  return stock

::method increaseStock                  /* increase stock of part */
  expose stock
  parse arg stockchange
  stock = stock + stockchange           /* - add change */
  return self~persistentUpdate         /* - store persistently */

::method decreaseStock                  /* decrease stock of part */
  expose stock
  parse arg stockchange
  if stockchange > stock then return -1 /* - check if possible */
  stock = stock - stockchange           /* - subtract change */
  return self~persistentUpdate         /* - store persistently */

::method detail                        /* return a detail line */
  expose partid description price stock
  return partid~right(5) ' ' description~left(15) ' ' ,
         price~right(8) ' ' stock~right(5)

::method makestring                    /* return default string */
  expose partid description
  return 'Part:' partid description

::method display                       /* display part data */
  expose partid description price stock
  say partid~right(5) ' ' description~left(15) ' ' ,
     price~right(8) ' ' stock~right(5)

```

Figure 121. (Part 3 of 3) *Base Part Class (base\part.ori)*

Base Part Class as Subclass of a SOM Class

```

/*-----*/
/* Base\part.som CarDealer - Part class (base/SOM)      ITS0-SJC */
/* (SOM part class, becomes carpart.cls)                */
/*-----*/

.local['Cardeal.Part.som'] = 'Yes'          /* mark as part in SOM */

::Class SOMPart EXTERNAL 'SOM Part'

::Class PartBase public subclass SOMPart

/*----- add OREXX class methods to the SOMObject Part -----*/

::method initialize class                      /* prepare class */
self~persistentLoad                          /* - load into memory */

/*----- instance methods -----*/

::method init                                /* initialize new part */
use arg partid, description, price, stock    /* - set all attributes */
self~set_pid(partid)
self~set_pprice(price)
self~set_pstock(stock)
self~set_pdesc(description)
if arg() \= 4 & arg() \= 5 then self~setnil
else self~class~add(self)                    /* - add to extent */
if arg() = 5 then self~persistentInsert       /* - real persistent new */

::method setnil private                      /* set part data nil */
self~class~remove(self)                     /* - remove from extent */
self~set_pid(0)
self~set_pprice(0)
self~set_pstock(0)
self~set_pdesc('none-')
return .nil

::method free                                /* free SOM storage */
self~class~remove(self)                     /* - remove from extent */
self~somFree

::method delete                              /* delete a part */
self~class~remove(self)                     /* - remove from extent */
self~setnil
self~persistentDelete                        /* - persistent delete */

::method increaseStock                       /* increase stock of part*/
parse arg stockchange
self~set_pstock(self~stock + stockchange)    /* - add the change */
return self~persistentUpdate                 /* - store persistently */

::method decreaseStock                       /* decrease stock of part*/
parse arg stockchange
if stockchange > self~stock then return -1    /* - check if enough */
self~set_pstock(self~stock - stockchange)    /* - subtract the change */
return self~persistentUpdate                 /* - store persistently */

::method makestring                          /* return default string */
return 'Part:' self~detail~substr(3,22)

```

Figure 122. Base Part Class as Subclass of a SOM Class
(base\part.som)

Persistent Class

```

/*-----*/
/* Base\persist.cls      CarDealer - Persistent class      ITS0-SJC */
/*-----*/

::class Persistent public mixinclass Object

/*----- class methods -----*/

::method persistentLoad class      /* default load into memory      */
  return 0

::method persistentStore class     /* default store back        */
  return 0

/*----- instance methods -----*/

::method persistentInsert          /* default new object        */
  return self~class~persistentStore

::method persistentDelete          /* default delete object     */
  return self~class~persistentStore

::method persistentUpdate          /* default update object     */
  return self~class~persistentStore

::method persistentInsertServ      /* new work-serv relation    */
  return self~class~persistentStore

::method persistentDeleteServ      /* delete work-serv relat.   */
  return self~class~persistentStore

```

Figure 123. *Persistent Class (base\persist.cls)*

Cardeal Class

```

/*-----*/
/* Base\cardeal.cls      CarDealer - Cardeal class      ITS0-SJC */
/*-----*/

.local['Cardeal.Cardeal.class'] = .Cardeal

::class Cardeal public

/*----- class methods -----*/

::method initialize class          /* prepare the class        */
  if RxFuncQuery('SysLoadFuncs') then do /* - load rexx utilities */
    call RxFuncAdd 'SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs'
  call SysLoadFuncs
end

```

Figure 124. (Part 1 of 2) *Cardeal Class (base\cardeal.cls)*

```

x = RxFuncDrop('mciRxInit') /* drop mciRx */ /* - drop multimedia funct.*/
self~mciRxInit /* - init multimedia funct.*/
.local['Cardeal.Part.class']~initialize /* - initialize all classes*/
.local['Cardeal.ServiceItem.class']~initialize
.local['Cardeal.Customer.class']~initialize
.local['Cardeal.Vehicle.class']~initialize
.local['Cardeal.WorkOrder.class']~initialize
return 0

::method terminate class /* application terminate */
if .local['Cardeal.Data.type'] = 'DB2' then do /* - check if DB2 */
call sqlexec "CONNECT RESET" /* - disconnect */
temp = value('TMP',, 'OS2ENVIRONMENT')
if temp = '' then temp = directory()
call SysFileTree temp"\t*.*", tempfiles, 'F0'
do i=1 to tempfiles.0 /* - erase temp files */
parse upper value substr(tempfiles.i,lastpos('\',tempfiles.i)+1) ,
with fn '.' fx
if pos('.'fx','.'BMP.WAV.AVI.')>0 then "@erase" tempfiles.i
end
end
if .local['Cardeal.Part.som'] = 'Yes' then do /* - check if SOM part */
do part1 over .local['Cardeal.Part.class']~extent
part1~free
end
.local['Cardeal.Part.class']~somUninit /* - uninitialized SOM */
end
do localx over .local~makearray /* - delete all local */
if localx~left(8) = 'Cardeal.' then .local~remove(localx)
end

::method playaudio class /* play an audio file */
arg filename MultiMedia
if filename = '' | MultiMedia = 0 then return
call mciRxSendString 'open waveaudio alias audio shareable wait', ,
'RetSt','0','0'
call mciRxSendString 'load audio' filename 'wait', 'RetSt','0','0'
call mciRxSendString 'set audio time format ms', 'RetSt','0','0'
call mciRxSendString 'play audio wait', 'RetSt','0','0'
call mciRxSendString 'close audio wait', 'RetSt','0','0'
call mciRxExit

::method playvideo class /* play a video file */
arg filename MultiMedia
if filename = '' | MultiMedia = 0 then return
call mciRxSendString 'open digitalvideo alias video shareable wait', ,
'RetSt','0','0'
call mciRxSendString 'load video' filename 'wait', 'RetSt','0','0'
call mciRxSendString 'set video time format ms', 'RetSt','0','0'
call mciRxSendString 'play video wait', 'RetSt','0','0'
call mciRxSendString 'close video wait', 'RetSt','0','0'
call mciRxExit

::method mciRxInit class private /* initialize multimedia */
expose done MultiMedia
if symbol('done') = 'VAR' then return MultiMedia
/* Load the DLL, initialize MCI REXX support */
if RxFuncQuery('mciRxInit') then do /* - load rex functions */
MultiMedia = (RXFUNCADD('mciRxInit','MCIAPI','mciRxInit')=0)
if MultiMedia then InitRC = mciRxInit()
end
else MultiMedia = 1
done = 1
return MultiMedia

```

Figure 124. (Part 2 of 2) Cardeal Class (base\cardeal.cls)

Persistence in Files

Configuration for File Storage

```

/*-----*/
/* FAT\carmodel.cfg      CarDealer - Model Config. (FAT)      ITS0-SJC */
/*-----*/

Parse source . . me .
maindir = me~left(me~lastpos('\')-1)      /* main cardeal directory */
if stream(maindir\base\cardeal.cls',c,'query exists') = '' then
    call carerror maindir

.local['Cardeal.Data.type'] = 'FAT'      /* Data in Files */
.local['Cardeal.Data.dir'] = maindir\FAT\Data'      /* Data directory */
.local['Cardeal.Media.dir'] = maindir\Media'      /* Media directory */

::requires 'base\cardeal.cls'

::requires 'fat\carcust.cls'
::requires 'fat\carvehi.cls'
::requires 'fat\carpart.cls'
::requires 'fat\carserv.cls'
::requires 'fat\carwork.cls'

```

Figure 125. *Configuration for File Storage (fat\carmodel.cfg)*

File Customer Class

```

/*-----*/
/* FAT\carcust.cls      CarDealer - Customer class (FAT)      ITS0-SJC */
/*-----*/

    .local['Cardeal.Customer.class'] = .Customer

::requires 'base\carcust.cls'
::requires 'base\persist.cls'

::class Customer public subclass CustomerBase inherit Persistent

/*----- class methods -----*/

::method persistentLoad class          /* load customers from file*/
    expose file
    file = .local['Cardeal.Data.dir']\customer.dat'
    call stream file, 'c', 'open read'
    do i = 0 by 1 while lines(file)      /* - read the file      */
        parse value linein(file) with customerNumber '9'x name '9'x address
        if left(customerNumber,2) = '/*' then iterate
        self~new(strip(customerNumber), strip(name), strip(address))
    end
    call stream file, 'c', 'close'
    return i

::method persistentStore class          /* store customers in file */
    expose file
    call stream file, 'c', 'open write replace'
    do custx over self~extent            /** - run over extent      */
        x = lineout(file, custx~fileFormat)
    end
    call stream file, 'c', 'close'
    return 0

/*----- instance methods -----*/

::method fileFormat                    /* prepare record for file */
    return strip(self~number)'9'x || left(self~name,20)'9'x || ,
        left(self~address,20)

```

Figure 126. *File Customer Class (fat\carcust.cls)*

File Vehicle Class

```

/*-----*/
/* FAT\carvehi.cls      CarDealer - Vehicle class (FAT)      ITS0-SJC */
/*-----*/

.local['Cardeal.Vehicle.class'] = .Vehicle

::requires 'base\carvehi.cls'
::requires 'base\persist.cls'

::class Vehicle public subclass VehicleBase inherit Persistent

/*----- class methods -----*/

::method persistentLoad class /* load vehicles from file */
  expose file
  file = .local['Cardeal.Data.dir']\vehicle.dat'
  custclass = .local['Cardeal.Customer.class']
  call stream file, 'c', 'open read'
  do i = 0 by 1 while lines(file) /* - read the file */
    parse value linein(file) ,
      with serialNumber '9'X make '9'X model '9'X year '9'X owner
    if left(serialNumber,2) = '/' then iterate
    self~new(strip(serialNumber), strip(make), strip(model), strip(year), ,
      custclass~findNumber(owner))
  end
  call stream file, 'c', 'close'
  return i

::method persistentStore class /* store vehicle in file */
  expose file
  call stream file, 'c', 'open write replace' /* - run over customers */
  do custx over .local['Cardeal.Customer.class']~extent
    do carx over custx~getVehicles /* - and their vehicles */
      x = lineout(file,carx~fileFormat)
    end
  end
  call stream file, 'c', 'close'
  return 0

/*----- instance methods -----*/

::method fileFormat /* prepare record for file */
  return strip(self~serial)'9'x || left(self~make,12)'9'x || ,
    left(self~model,10)'9'x || strip(self~year)'9'x || ,
    strip(self~getowner~number)

::method getmedianumber /* return number of media */
  expose medianumber mediacontrol picfile
  if symbol('medianumber') = 'VAR' then return medianumber
  medianumber = 0
  mediacontrol = ''
  picfile = .array~new
  mediafile = .local['Cardeal.Media.dir']\media.dat'
  do i=1 by 1 while lines(mediafile)>0 /* - read media controlfile*/
    line = linein(mediafile)
    if left(line,2) = '/' then iterate
    parse var line serial ',' title ',' file /* - check for serial */
    if self~serial = strip(serial) then do
      medianumber = medianumber + 1
      picfile[medianumber] = strip(file) /* - build control info */
      mediacontrol = mediacontrol||left(strip(title),20),file ';'
    end
  end
  end
end

```

Figure 127. (Part 1 of 2) File Vehicle Class (fat\carvehi.cls)


```

x = stream(mediafile,'c','close')
return medianumber

::method getmediacontrol          /* return media controlinfo*/
  expose medianumber mediacontrol
  if symbol("medianumber") = 'LIT' then return ''
  return mediacontrol

::method getmediainfo            /* return a media file      */
  expose medianumber mediacontrol picfile
  if symbol("medianumber") = 'LIT' then return ''
  arg medianum
  if medianumber = 0 | mediacontrol = '' | ,
    medianum > medianumber | medianum <= 0 then return ''
  mediatitle = substr(mediacontrol,medianum*30-29,20)
  vfacts = .local['Cardeal.Media.dir']\'\'picfile[medianum]
  if mediatitle = 'Fact-sheet' then do
    factdata = linein(vfacts)
    x = stream(vfacts,'c','close')
    vfacts = factdata
  end
  return mediatitle::'vfacts

```

Figure 127. (Part 2 of 2) *File Vehicle Class (fat\carvehi.cls)*

File Work Order Class

```

/*-----*/
/* FAT\carwork.cls    CarDealer - WorkOrder class (FAT)    ITS0-SJC */
/*-----*/

.local['Cardeal.WorkOrder.class'] = .WorkOrder

::requires 'base\carwork.cls'
::requires 'base\persist.cls'

::class WorkOrder public subclass WorkOrderBase inherit Persistent

/*----- class methods -----*/

::method persistentLoad class /* load work orders file */
  expose file
  file = .local['Cardeal.Data.dir']\workord.dat'
  custclass = .local['Cardeal.Customer.class']
  servclass = .local['Cardeal.ServiceItem.class']
  call stream file, 'c', 'open read'
  do i = 0 by 1 while lines(file) /* - read the file */
    parse value linein(file) with orderno '9'x date '9'x cost ,
      '9'x status '9'x owner '9'x car '9'x items
    if left(orderno,2) = '/' then iterate
    custx = custclass~findNumber(owner) /* -create new work order */
    wo = self~new(strip(orderno), strip(date), strip(cost), ,
      strip(status), custx, custx~findVehicle(car))
    do while items \= '' /* -add services to order */
      parse var items itemx '9'x items
      wo~addServiceItem(servclass~findNumber(itemx))
    end
  end
  call stream file, 'c', 'close'
  return i

::method persistentStore class /* store workorders in file*/
  expose file
  call stream file, 'c', 'open write replace'
  do ordx over self~extent /* - run over extent */
    x = lineout(file,ordx~fileFormat)
  end
  call stream file, 'c', 'close'
  return 0

/*----- instance methods -----*/

::method fileFormat /* prepare record for file */
  out = strip(self~number)'9'x || strip(self~date)'9'x || ,
    strip(self~cost)'9'x || strip(self~getstatus)'9'x || ,
    strip(self~getcustomer~number)'9'x || strip(self~getvehicle~serial)
  workserv = self~class~getWorkServRel
  do servx over workserv~allat(self)
    out = out'9'x || servx~number
  end
  return out

```

Figure 128. *File Work Order Class (fat\carwork.cls)*

File Service Item Class

```

/*-----*/
/* FAT\carserv.cls      CarDealer - ServiceItem class (FAT) ITS0-SJC */
/*-----*/

.local['Cardeal.ServiceItem.class'] = .ServiceItem

::requires 'base\carserv.cls'
::requires 'base\persist.cls'

::class ServiceItem public subclass ServiceItemBase inherit Persistent

/*----- class methods -----*/

::method persistentLoad class          /* load service from file */
  expose file
  file = .local['Cardeal.Data.dir']\service.dat'
  partclass = .local['Cardeal.Part.class']
  call stream file, 'c', 'open read'
  do i = 0 by 1 while lines(file)      /* - read the file */
    parse value linein(file) with ,
      itemNumber '9'x description '9'x laborCost '9'x parts
    if left(itemNumber,2) = '/' then iterate
    si = self~new(strip(itemNumber), strip(description), strip(laborCost))
    do while parts \= ''                /* - add parts to service */
      parse var parts partnum '9'x quant '9'x parts
      si~usesPart(partclass~findNumber(partnum), strip(quant))
    end
  end
  call stream file, 'c', 'close'
  return i

::method persistentStore class          /* store services in file */
  /* no change in data ever */
  return 0

/*----- instance methods -----*/

::method fileFormat                    /* prepare record for file */
  /* never used since service items are not updated */
  out = strip(self~number)'9'x || left(self~description,20)'9'x || ,
    strip(self~laborcost)
  do partx over parts
    out = out'9'x || right(partx~number,2) || ,
      '9'x || right(self~getquantity(partx),2)
  end
  return out

```

Figure 129. *File Service Item Class (fat\carserv.cls)*

File Part Class

```

/*-----*/
/* FAT\carpart.cls      CarDealer - Part class (FAT)      ITS0-SJC */
/*-----*/

.local['Cardeal.Part.class'] = .Part

::requires 'base\carpart.cls'
::requires 'base\persist.cls'

::class Part public subclass PartBase inherit Persistent

/*----- class methods -----*/

::method persistentLoad class /* load parts from file */
  expose file
  file = .local['Cardeal.Data.dir']\part.dat'
  call stream file, 'c', 'open read'
  do i = 0 by 1 while lines(file) /* - read the file */
    parse value linein(file) with ,
      partid '9'x description '9'x price '9'x stock
    if left(partid,2) = '/' then iterate
    self~new(strip(partid), strip(description), strip(price), strip(stock))
  end
  call stream file, 'c', 'close'
  return i

::method persistentStore class /* store parts in file */
  expose file
  call stream file, 'c', 'open write replace'
  do partx over self~extent /* - run over extent */
    x = lineout(file, partx~fileFormat)
  end
  call stream file, 'c', 'close'
  return 0

/*----- instance methods -----*/

::method fileFormat /* prepare record for file */
  return strip(self~number)'9'x || left(self~description,15)'9'x || ,
    strip(self~price)'9'x || strip(self~stock)

```

Figure 130. *File Part Class (fat\carpart.cls)*

Persistence in DB2

Configuration for DB2 Storage

```

/*-----*/
/* DB2\carmodel.cfg  CarDealer - Model Config. (DB2)      ITS0-SJC */
/*-----*/

Parse source . . me .
maindir = me~left(me~lastpos('\')-1)      /* main cardeal directory */
if stream(maindir\base\cardeal.cls',c,'query exists') = '' then
    call carerror maindir

call rxftsql                                /* Rexx DB2 functions      */
call sqlexec "CONNECT RESET"
call sqlexec "CONNECT TO DEALERDB"        /* connect to database      */
if sqlca.sqlcode \= 0 then do; say 'Cannot connect to DEALERDB'
                                exit 16; end

.local['Cardeal.Data.type'] = 'DB2'        /* Data in DB2              */
.local['Cardeal.Data.dir'] = '-none-'      /* Data in DB2              */
.local['Cardeal.Media.dir'] = '-none-'     /* Media in DB2             */

::requires 'base\cardeal.cls'

::requires 'db2\carcust.cls'
::requires 'db2\carvehi.cls'
::requires 'db2\carpart.cls'
::requires 'db2\carserv.cls'
::requires 'db2\carwork.cls'

```

Figure 131. Configuration for DB2 Storage (*db2\carmodel.cfg*)

DB2 Customer Class

```

/*-----*/
/* DB2\carcust.cls  CarDealer - Customer class (DB2)      ITS0-SJC */
/*-----*/

.local['Cardeal.Customer.class'] = .Customer

::requires 'base\carcust.cls'

::class Customer public subclass CustomerBase

/*----- class methods -----*/

::method persistentLoad class                /* null, load by demand */
    return 0

::method findNumber class                    /* load customer by number */
    use arg custnum

```

Figure 132. (Part 1 of 3) DB2 Customer Class (*db2\carcust.cls*)

```

vehicclass = .local['Cardeal.Vehicle.class']
workclass = .local['Cardeal.WorkOrder.class']
custx = self~findNumber:super(custnum) /* - check if in memory */
if custx \= .nil then return custx
stmt = 'select c.custname, c.custaddr' ,
      '  from cardeal.customer c' ,
      '  where c.custnum = ' custnum
call sqlxec 'PREPARE s1 FROM :stmt'
call sqlxec 'DECLARE c1 CURSOR FOR s1'
call sqlxec 'OPEN c1'
call sqlxec 'FETCH c1 INTO :xcustn, :xcusta'
if sqlca.sqlcode = 0 then do
  custx = self~new(custnum, xcustn, xcusta)
  vehicclass~persistentLoadByCust(custx) /* - load vehicles of cust.*/
  workclass~persistentLoadByCust(custx) /* - load workorders */
end
else custx = .nil
call sqlxec 'CLOSE c1'
return custx

::method findName class /* find customer by name */
use arg custsearch
custnames = .list~new /* - prepare result list */
stmt = "select c.custnum, c.custname, c.custaddr" ,
      "  from cardeal.customer c" ,
      "  where c.custname like ? order by 2"
call sqlxec 'PREPARE s1 FROM :stmt'
call sqlxec 'DECLARE c1 CURSOR FOR s1'
xsearch = ""custsearch%"
call sqlxec "OPEN c1 USING :xsearch"
do icust=0 by 1 until rcc \= 0 /* - search table with LIKE*/
  call sqlxec 'FETCH c1 INTO :xcustno, :xcustn, :xcusta'
  rcc = sqlca.sqlcode
  if rcc = 0 then do
    custstring = xcustno~right(3)~'xcustn'~'xcusta'
    custnames~insert(custstring)
  end
end
call sqlxec 'CLOSE c1'
return custnames~makearray /* - return result array */

::method findAddress class /* find customer by address*/
use arg custsearch
stmt = "select c.custnum, c.custname, c.custaddr" ,
      "  from cardeal.customer c" ,
      "  where c.custaddr = ?"
call sqlxec 'PREPARE s1 FROM :stmt'
call sqlxec 'DECLARE c1 CURSOR FOR s1'
xsearch = ""custsearch%"
call sqlxec "OPEN c1 USING :xsearch"
  call sqlxec 'FETCH c1 INTO :xcustno, :xcustn, :xcusta'
  rcc = sqlca.sqlcode
call sqlxec 'CLOSE c1'
if rcc = 0 then return xcustno /* return customer number */
else return ''

```

Figure 132. (Part 2 of 3) DB2 Customer Class (db2\carcust.cls)

```

/*----- instance methods -----*/
::method persistentInsert                                /* store new customer */
    insertstmt = "insert into cardeal.customer",
                " values('self~number','self~name','self~address')"
    call sqlxec 'EXECUTE IMMEDIATE :insertstmt'
    if sqlca.sqlcode \= 0 then do
        say 'cust insert' sqlca.sqlcode sqlmsg
        self~setnil
    end
    else call sqlxec 'COMMIT'
    return sqlca.sqlcode

::method persistentUpdate                                /* update a customer */
    updatetstmt = "update cardeal.customer",
                  " set custname = 'self~name', custaddr = 'self~address'",
                  " where custnum = 'self~number'"
    call sqlxec 'EXECUTE IMMEDIATE :updatetstmt'
    if sqlca.sqlcode \= 0 then say 'customer update' sqlca.sqlcode sqlmsg
    else call sqlxec 'COMMIT'
    return sqlca.sqlcode

::method persistentDelete                                /* delete a customer */
    delstmt = 'delete from cardeal.customer where custnum = ' self~number
    call sqlxec 'EXECUTE IMMEDIATE :delstmt'
    if sqlca.sqlcode \= 0 then say 'cust delete' sqlca.sqlcode sqlmsg
    else call sqlxec 'COMMIT'
    return sqlca.sqlcode

```

Figure 132. (Part 3 of 3) DB2 Customer Class (db2\carcust.cls)

DB2 Vehicle Class

```

/*-----*/
/* DB2\carvehi.cls      CarDealer - Vehicle class (DB2)      ITS0-SJC */
/*-----*/

    .local['Cardeal.Vehicle.class'] = .Vehicle

::requires 'base\carvehi.cls'

::class Vehicle public subclass VehicleBase

/*----- class methods -----*/

::method persistentLoad class                                /* null, load by demand */
    return 0
::method persistentLoadByCust class                            /* load vehicle of customer*/
    use arg custx
    customerNumber = custx~number

```

Figure 133. (Part 1 of 3) DB2 Vehicle Class (db2\carvehi.cls)

```

stmt = 'select v.serialnum, v.make, v.model, v.year'
      'from cardeal.vehicle v where v.custnum = ' customerNumber
call sqlxec 'PREPARE s2 FROM :stmt'
call sqlxec 'DECLARE c2 CURSOR FOR s2'
call sqlxec 'OPEN c2'
do until rcv \= 0
    call sqlxec 'FETCH c2 INTO :xserial, :xmake, :xmodel, :xyear'
    rcv = sqlca.sqlcode
    if rcv = 0 then
        carx = self~new(xserial, xmake, xmodel, xyear, custx)
    end
call sqlxec 'CLOSE c2'
return 0

/*----- instance methods -----*/
::method persistentInsert /* store new vehicle */
    custnum = self~getowner~number
    insertstmt = "insert into cardeal.vehicle" ,
                " (serialnum, custnum, make, model, year)" ,
                " values("self~serial", "custnum", "self~make", " ,
                " "self~model", "self~year")"
    /* say 'created' self 'in DB2' */
    call sqlxec 'EXECUTE IMMEDIATE :insertstmt'
    if sqlca.sqlcode \= 0 then do
        say 'vehicle insert' sqlca.sqlcode sqlca.sqlerrmc
        self~setnil
    end
    call sqlxec 'COMMIT'

::method persistentUpdate /* update vehicle data */
    updatetstmt = "update cardeal.vehicle" ,
                 " set make = "self~make", model = "self~model", " ,
                 " year = " selfyear ,
                 " where serialnum = " self~serial
    call sqlxec 'EXECUTE IMMEDIATE :updatetstmt'
    if sqlca.sqlcode \= 0 then say 'customer update' sqlca.sqlcode sqlmsg
    else call sqlxec 'COMMIT'
    return sqlca.sqlcode

::method persistentDelete /* delete a vehicle */
    delstmt = 'delete from cardeal.vehicle where serialnum = ' self~serial
    call sqlxec 'EXECUTE IMMEDIATE :delstmt'
    if sqlca.sqlcode \= 0 then say 'vehicle delete' sqlca.sqlcode sqlmsg
    else call sqlxec 'COMMIT'
    return sqlca.sqlcode

::method getmedianumber /* number of media files */
    expose medianumber mediacontrol /* - in the BLOB */
    if symbol("medianumber") = 'VAR' then return medianumber
    medianumber = 0
    mediacontrol = ''
    stmt = 'select substr(v.pictures,1,3)' ,
          'from cardeal.vehicle v where v.serialnum = ' self~serial
    call sqlxec 'PREPARE s2 FROM :stmt'
    if sqlca.sqlcode \= 0 then return 0
    vpicind = -1
    call sqlxec 'DECLARE c2 CURSOR FOR s2'
    call sqlxec 'OPEN c2'
    call sqlxec 'FETCH c2 INTO :vpic :vpicind'
    call sqlxec 'CLOSE c2'
    if vpicind >= 0 then medianumber = vpic
    return medianumber

```

Figure 133. (Part 2 of 3) DB2 Vehicle Class (db2\carvehi.cls)


```

::method getmediacontrol                                  /* return media controlinfo*/
  expose medianumber mediacontrol
  if symbol("medianumber") = 'LIT' then return
  if medianumber <= 0 then return
  stmt = 'select substr(v.pictures,5,30*'medianumber')',
        ' from cardeal.vehicle v where v.serialnum = ' self~serial
  call sqlxec 'PREPARE s2 FROM :stmt'
  call sqlxec 'DECLARE c2 CURSOR FOR s2'
  call sqlxec 'OPEN c2'
    call sqlxec 'FETCH c2 INTO :vpic :vpicind'
    rcv = sqlca.sqlcode
  call sqlxec 'CLOSE c2'
  if rcv = 0 & vplicind >= 0 then mediacontrol = vplic
  return mediacontrol

::method getmediainfo                                    /* return one media file */
  expose medianumber mediacontrol
  parse source env .
  if env = 'OS/2' then env = 'OS2ENVIRONMENT'
                        else env = 'ENVIRONMENT'
  if symbol("medianumber") = 'LIT' then return
  if mediacontrol = '' then self~getmediacontrol
  arg medianum
  if medianumber = 0 | medianum > medianumber | medianum <= 0 | ,
    mediacontrol = '' then return
  mediatitle = substr(mediacontrol,medianum*30-29,20)
  medialength = substr(mediacontrol,medianum*30- 8, 8)
  mediastart = 7 + 30 * medianumber
  do i=1 to medianum -1
    blg = substr(mediacontrol,i*30-8,8)
    mediastart = mediastart + blg
  end
  call sqlxec 'CLEAR SQL VARIABLE DECLARATIONS'
  call sqlxec 'DECLARE :vpic3 LANGUAGE TYPE BLOB FILE'
  vpic3.file_options = 'OVERWRITE'
  temp = value('TMP',,env); if temp = '' then temp = directory()
  tnam = 't'self~serial'medianum
  select
    when mediatitle = 'Fact-sheet' then vpic3.name = ''
    when mediatitle = 'Audio'   then vpic3.name = temp'\tnam'.WAV'
    when mediatitle = 'Video'   then vpic3.name = temp'\tnam'.AVI'
    otherwise                    vpic3.name = temp'\tnam'.BMP'
  end
  vfacts = vpic3.name
  stmt = 'select substr(v.pictures,'mediastart','medialength')',
        ' from cardeal.vehicle v where v.serialnum = ' self~serial
  call sqlxec 'PREPARE s2 FROM :stmt'
  call sqlxec 'DECLARE c2 CURSOR FOR s2'
  call sqlxec 'OPEN c2'
    if vfacts = '' then call sqlxec 'FETCH c2 INTO :vfacts'
                        else call sqlxec 'FETCH c2 INTO :vpic3 :vpicind3'
    if sqlca.sqlcode \= 0 then vfacts = ''
  call sqlxec 'CLOSE c2'
  call sqlxec 'CLEAR SQL VARIABLE DECLARATIONS'
  return mediatitle::'vfacts

```

Figure 133. (Part 3 of 3) DB2 Vehicle Class (db2\carvehi.cls)

DB2 Work Order Class

```

/*-----*/
/* DB2\carwork.cls   CarDealer - WorkOrder class (DB2)   ITS0-SJC */
/*-----*/

.local['Cardeal.WorkOrder.class'] = .WorkOrder

::requires 'base\carwork.cls'

::class WorkOrder public subclass WorkOrderBase

/*----- class methods -----*/

::method persistentLoad class /* null, load by demand */
return 0

::method findNumber class /* find workorder by number*/
use arg worknum
custclass = .local['Cardeal.Customer.class']
workx = self~findNumber:super(worknum) /* - check in memory first */
if workx \= .nil then return workx /* - return if found */
stmt = 'select w.custnum'
      ' from cardeal.workorder w where w.ordernum = ' worknum
call sqlexec 'PREPARE s3 FROM :stmt'
call sqlexec 'DECLARE c3 CURSOR FOR s3'
call sqlexec 'OPEN c3'
call sqlexec 'FETCH c3 INTO :xcustnum'
rcw = sqlca.sqlcode
call sqlexec 'CLOSE c3'
if rcw = 0 then do
  custx = custclass~findNumber(xcustnum)
  if custx \= .nil then
    do workx over self~extent
      if workx~number = worknum then return workx
    end
  end
return .nil

::method findStatus class /* find workorder by status*/
use arg xstatus
worklist = .list~new /* - prepare result list */
stmt = 'select w.ordernum, w.orderdate, w.cost, w.status, '
      ' c.custname, v.make, v.model'
      ' from cardeal.workorder w, cardeal.customer c, cardeal.vehicle v '
      ' where w.custnum = c.custnum and w.serialnum = v.serialnum '
      ' and w.status in (?, ?)'
      ' order by 1'
hostvar = ':xordno, :xdate, :xcost, :xstatus, :xcustn, :xmake, :xmodel'
call sqlexec 'PREPARE s3 FROM :stmt'
call sqlexec 'DECLARE c3 CURSOR FOR s3'
xstat1 = 0
xstat2 = 1
if xstatus = 0 then xstat2=0
if xstatus = 1 then xstat1=1
call sqlexec 'OPEN c3 USING :xstat1, :xstat2'
do iwork = 0 by -1 until rcw \= 0
  call sqlexec 'FETCH c3 INTO' hostvar
  rcw = sqlca.sqlcode
  if rcw = 0 then do
    if xstatus = 0 then statusx = 'Incomplete'
    else statusx = 'Complete'
    workstring = xordno~left(3) ' ' xdate xcost~right(6) statusx~left(11)
    (xmake~strip-'xmodel~strip)~left(20) xcustn
    worklist~insert(workstring,.nil)
  end
end
call sqlexec 'CLOSE c3'
return worklist~makearray

```

Figure 134. (Part 1 of 3) DB2 Work Order Class (db2\carwork.cls)

```

::method newNumber class                                /* create new order number */
  stmt = 'select max(ordernum) from cardeal.workorder'
  call sqlxec 'PREPARE s3 FROM :stmt'
  call sqlxec 'DECLARE c3 CURSOR FOR s3'
  call sqlxec 'OPEN c3'
  call sqlxec 'FETCH c3 INTO :xmax'
  call sqlxec 'CLOSE c3'
  return xmax + 1

::method persistentLoadByCust class                    /* load workorders of cust.*/
  use arg custx
  servclass = .local['Cardeal.ServiceItem.class']
  customerNumber = custx~number
  stmt = 'select w.ordernum, w.cost, w.orderdate, w.status, w.serialnum',
        ' from cardeal.workorder w where w.custnum = ' customerNumber
  call sqlxec 'PREPARE s4 FROM :stmt'
  call sqlxec 'DECLARE c4 CURSOR FOR s4'
  call sqlxec 'OPEN c4'
  do until rcw \= 0                                     /* - run over orders */
    call sqlxec 'FETCH c4 INTO :xorder, :xcost, :xdate, :xstatus, :xserial'
    rcw = sqlca.sqlcode
    if rcw = 0 then do
      cars = custx~getVehicles
      do carx over cars                                  /* - find matching car */
        if carx~serial = xserial then do                /* for work order */
          orderx = self~new(xorder, xdate, xcost, xstatus, custx, carx)
          servitems = servclass~extent
          stmt2 = 'select r.itemnum',
                  ' from cardeal.workserv r where r.ordernum = ' xorder
          call sqlxec 'PREPARE s5 FROM :stmt2'
          call sqlxec 'DECLARE c5 CURSOR FOR s5'
          call sqlxec 'OPEN c5'
          do until rcs \= 0                               /* - and add rels to serv, */
            call sqlxec 'FETCH c5 INTO :xitem'
            rcs = sqlca.sqlcode
            if rcs = 0 then
              do servx over servitems
                if servx~number = xitem then
                  orderx~addServiceItem(servx)
              end
            end
            call sqlxec 'CLOSE c5'
          end
          end /*cars*/
        end /*rcw=0*/
      end
    call sqlxec 'CLOSE c4'
    return 0

/*----- instance methods -----*/

::method persistentInsert                              /* store new work order */
  custnum = self~getcustomer~number
  carserial = self~getvehicle~serial
  insertstmt = "insert into cardeal.workorder"
              " values('self~number','custnum','carserial','" ,
              " self~cost','self~date','self~getstatus')"
  call sqlxec 'EXECUTE IMMEDIATE :insertstmt'
  if sqlca.sqlcode \= 0 then do
    say 'workorder insert' sqlca.sqlcode sqlmsg
    self~setnil
  end
  else call sqlxec 'COMMIT'
  return sqlca.sqlcode

```

Figure 134. (Part 2 of 3) DB2 Work Order Class (db2\carwork.cls)

```

::method persistentDelete                                /* delete work order */
    delstmt = 'delete from cardeal.workorder where ordernum =' self~number
    call sqlexec 'EXECUTE IMMEDIATE :delstmt'
    if sqlca.sqlcode \= 0 then say 'order delete' sqlca.sqlcode sqlca.sqlerrmc
    delstmt = 'delete from cardeal.workserv where ordernum =' self~number
    call sqlexec 'EXECUTE IMMEDIATE :delstmt'
    if sqlca.sqlcode \= 0 & sqlca.sqlcode \= 100 then
        say 'order-serv delete' sqlca.sqlcode sqlmsg
    else call sqlexec 'COMMIT'
    return sqlca.sqlcode

::method persistentInsertServ                            /* add service item */
    use arg itemnum
    if sqlca.sqlcode \= 0 then say 'workserv delete' sqlca.sqlcode sqlca.sqlerrmc
    insertstmt = 'insert into cardeal.workserv values('self~number',' itemnum')'
    call sqlexec 'EXECUTE IMMEDIATE :insertstmt'
    rci = sqlca.sqlcode
    if rci \= 0 then say 'workserv insert' sqlca.sqlcode sqlmsg
    else call sqlexec 'COMMIT'
    return sqlca.sqlcode

::method persistentDeleteServ                            /* delete service item */
    use arg itemnum
    deletestmt = 'delete from cardeal.workserv',
                ' where ordernum =' self~number 'and itemnum =' itemnum
    call sqlexec 'EXECUTE IMMEDIATE :deletestmt'
    if sqlca.sqlcode \= 0 then say 'workserv delete' sqlca.sqlcode sqlerrmc
    else call sqlexec 'COMMIT'
    return sqlca.sqlcode

::method persistentUpdate                                /* update work order data*/
    updatestmt = 'update cardeal.workorder',
                ' set cost =' self~cost', status =' self~getstatus ,
                ' where ordernum =' self~number
    call sqlexec 'EXECUTE IMMEDIATE :updatestmt'
    if sqlca.sqlcode \= 0 then say 'workorder update' sqlca.sqlcode sqlmsg
    else call sqlexec 'COMMIT'
    return sqlca.sqlcode

```

Figure 134. (Part 3 of 3) DB2 Work Order Class (db2\carwork.cls)

DB2 Service Item Class

```

/*-----*/
/* DB2\carserv.cls      CarDealer - ServiceItem class (DB2) ITS0-SJC */
/*-----*/

.local['Cardeal.ServiceItem.class'] = .ServiceItem

::requires 'base\carserv.cls'

::class ServiceItem public subclass ServiceItemBase

/*----- class methods -----*/

::method persistentLoad class /* load all service items */
partclass = .local['Cardeal.Part.class']
stmt = 'select s.itemnum, s.labor, s.description',
      ' from cardeal.service s',
      ' order by 1'
hostvar = ':xitem, :xlabor, :xdesc1'
call sqlexec 'PREPARE s1 FROM :stmt'
if sqlca.sqlcode \= 0 then
  say 'sqlerror service items prepare:' sqlca.sqlcode sqlmsg
call sqlexec 'DECLARE c1 CURSOR FOR s1'
call sqlexec 'OPEN c1'
if sqlca.sqlcode \= 0 then
  say 'sqlerror service items open:' sqlca.sqlcode sqlmsg
do iserv = 0 by 1 until sqlca.sqlcode \= 0 /* - run over service table */
call sqlexec 'FETCH c1 INTO' hostvar
if sqlca.sqlcode \= 0 & sqlca.sqlcode \= 100 then
  say 'sqlerror service items fetch:' sqlca.sqlcode sqlmsg
else if sqlca.sqlcode = 0 then do
  /* say 'creating service item' xitem */
  servx = self~findNumber(xitem)
  if servx = .nil then
    servx = self~new(xitem, xdesc1, xlabor)
  end
end
call sqlexec 'CLOSE c1'
if sqlca.sqlcode \= 0 then
  say 'sqlerror service items close:' sqlca.sqlcode sqlmsg
/* say 'Loaded' self~gettextent~items 'service items' */
/* - add service-part rels */
stmt = 'select r.itemnum, r.quantity, r.partnum',
      ' from cardeal.servpart r'
hostvar = ':xitem, :xquan, :xpartid'
call sqlexec 'PREPARE s1 FROM :stmt'
if sqlca.sqlcode \= 0 then
  say 'sqlerror service-parts prepare:' sqlca.sqlcode sqlmsg
call sqlexec 'DECLARE c1 CURSOR FOR s1'
call sqlexec 'OPEN c1'
if sqlca.sqlcode \= 0 then
  say 'sqlerror service-parts open:' sqlca.sqlcode sqlmsg
do iservprt = 0 by 1 until sqlca.sqlcode \= 0 /* - run over servpart tab.*/
call sqlexec 'FETCH c1 INTO' hostvar
if sqlca.sqlcode \= 0 & sqlca.sqlcode \= 100 then
  say 'sqlerror service-parts fetch:' sqlca.sqlcode sqlmsg
else if sqlca.sqlcode = 0 then do
  /* say 'creating service-part' xitem xpartid */
  partx = partclass~findNumber(xpartid)

```

Figure 135. (Part 1 of 2) DB2 Service Item Class (db2\carserv.cls)

```

        if partx = .nil then
            say 'Service item' xitem 'uses non-existing part' xpartid
            servx = self~findNumber(xitem)
            if servx = .nil then
                say 'Service item' xitem 'not in service table'
            else
                servx~usesPart(partx, xquan)
            end
        end
    end
    call sqlexec 'CLOSE c1'
    if sqlca.sqlcode \= 0 then
        say 'sqlerror service-parts close:' sqlca.sqlcode sqlmsg
    /* say 'Loaded' partclass~getextent~items 'parts' */
    /* say 'Loaded' iservprt 'service/part relationships' */
    /* say 'All sample data read' */
    return iserv

```

Figure 135. (Part 2 of 2) DB2 Service Item Class (db2\carserv.cls)

DB2 Part Class

```

/*-----*/
/* DB2\carpart.cls      CarDealer - Part class (DB2)      ITS0-SJC */
/*-----*/

.local['Cardeal.Part.class'] = .Part

::requires 'base\carpart.cls'

::class Part public subclass PartBase

/*----- class methods -----*/

::method persistentLoad class /* load all parts from DB2 */
    stmt = 'select p.partnum, p.price, p.stock, p.description' ,
           ' from cardeal.part p' ,
           ' order by 1'
    hostvar = ':xpartid, :xprice, :xstock, :xdesc2'
    call sqlexec 'PREPARE s1 FROM :stmt'
    if sqlca.sqlcode \= 0 then
        say 'sqlerror parts prepare:' sqlca.sqlcode sqlmsg
    call sqlexec 'DECLARE c1 CURSOR FOR s1'
    call sqlexec 'OPEN c1'
    if sqlca.sqlcode \= 0 then
        say 'sqlerror parts open:' sqlca.sqlcode sqlmsg
    do ipart = 0 by 1 until sqlca.sqlcode \= 0 /* - run over all parts */
    call sqlexec 'FETCH c1 INTO' hostvar
    if sqlca.sqlcode \= 0 & sqlca.sqlcode \= 100 then
        say 'sqlerror parts fetch:' sqlca.sqlcode sqlmsg
    else if sqlca.sqlcode = 0 then do
        partx = self~findNumber(xpartid)
        if partx = .nil then
            partx = self~new(xpartid, xdesc2, xprice, xstock)
        end
    end
    call sqlexec 'CLOSE c1'
    if sqlca.sqlcode \= 0 then
        say 'sqlerror parts close:' sqlca.sqlcode sqlmsg
    return ipart

```

Figure 136. (Part 1 of 2) DB2 Part Class (db2\carpart.cls)

```

/*----- instance methods -----*/

::method persistentUpdate                                /* update a part */
    use arg quant
    updatetxt = 'update cardeal.part set stock = ' self~stock ,
               ' where partnum = ' self~number
    call sqlxec 'EXECUTE IMMEDIATE :updatetxt'
    if sqlca.sqlcode \= 0 then say 'part-update' sqlca.sqlcode sqlmsg
    else call sqlxec 'COMMIT'
    return sqlca.sqlcode

::method persistentInsert                                /* store new part */
    inserttxt = "insert into cardeal.part" ,
               " values('self~number','self~price','self~stock','" ,
               " 'self~description'" )
    call sqlxec 'EXECUTE IMMEDIATE :inserttxt'
    if sqlca.sqlcode \= 0 then say 'part-insert' sqlca.sqlcode sqlmsg
    else call sqlxec 'COMMIT'
    return sqlca.sqlcode

```

Figure 136. (Part 2 of 2) DB2 Part Class (db2\carpart.cls)

Running the Car Dealer Application

Program to Run the Car Dealer Application

```

/*-----*/
/* car-run.cmd          CarDealer - Run Car Dealer          ITS0-SJC */
/*                      (AUI or GUI, File or DB2, optional SOM) */
/*-----*/

parse source . . me .
sourcdir = me~left(me~lastpos('\')-1)
curdir = directory()      /* save current directory */
new = directory(sourcdir) /* make CARDEAL current directory */
env = 'OS2ENVIRONMENT'

arg p1 p2 p3 '(' quiet
if left(strip(quiet),1) = 'Q' then talk = 0
else talk = 1

```

Figure 137. (Part 1 of 2) Command to Run the Car Dealer (\car-run.cmd)

```

if p1 = '' | p1 = '?' then do
  say 'Syntax: CAR-RUN [F | D | R] ' ,
    '[O | S] ' ,
    '[A | G | P | X]'
  say ' first parm: F = File, D = DB2/2, R = RAM (memory) only'
  say ' second parm: O = OREXX Part class(default), S = SOM Part class'
  say ' third parm: A = Ascii window, G = DrRexx, P = VisProRexx, X = Vx-Rexx'
  say ' parameters : in any sequence, blank separated'
  say '                setup F|D|M is saved, O|S is saved'
  return
end

opt = left(strip(p1),1) left(strip(p2),1) left(strip(p3),1)

/* setup data storage */

select
  when pos('F',opt)>0 then do; "@copy FAT\carmodel.cfg >null"
    if talk then say 'Setup for FAT data'; end
  when pos('D',opt)>0 then do; "@copy DB2\carmodel.cfg >null"
    if talk then say 'Setup for DB2 data'; end
  when pos('R',opt)>0 then do; "@copy RAM\carmodel.cfg >null"
    if talk then say 'Setup for Memory data'; end
  otherwise nop
end

/* setup if SOM is used or not */

select
  when pos('O',opt)>0 then do; "@copy Base\part.ori Base\carpart.cls >null"
    if talk then say 'Setup for ORExx part class'; end
  when pos('S',opt)>0 then do; "@copy Base\part.som Base\carpart.cls >null"
    if talk then say 'Setup for SOM part class'; end
  otherwise nop
end

/* Run program in AUI or GUI mode */

select
  when pos('A',opt)>0 then call "AUI\car-aui"
  when pos('G',opt)>0 then "DrDialCD\car-gui.exe"
  when pos('P',opt)>0 then "VisProCD\car-gui.exe"
  when pos('X',opt)>0 then do
    vxdir = directory(sourcedir'\VxRexxCd')
    p = value('PATH',,env)
    if pos(sourcedir';',p) = 0 then p = value('PATH',p;','sourcedir';',env)
    "car-gui.exe"
  end
  otherwise
    if talk then say 'You can now run any Car Dealer application (ASCII or GUI)'
end

curdir = directory(curdir) /* restore current directory */
return

```

Figure 137. (Part 2 of 2) Command to Run the Car Dealer (\car-run.cmd)

B

Definition for Syntax Diagram Structure

Throughout this book, syntax is described using the structure defined below:

- Syntax diagrams are read from left to right, top to bottom, following the path of the line.

The  symbol indicates the beginning of a statement.

The  symbol indicates that the statement syntax is continued on the next line.

The  symbol indicates that a statement is continued from the previous line.

The  symbol indicates the end of a statement.

Diagrams of syntactical units other than complete statements start with the  symbol and end with the  symbol.

- ❑ Required items appear on the horizontal line (the main path).

➡➡ STATEMENT — *required_item* ————— ➡➡

- ❑ Optional items appear below the main path.

➡➡ STATEMENT ————
 └ *optional_item* ┘ ————— ➡➡

- ❑ Choices appear vertically, in a stack. If one item *must* be chosen, it will appear on the main path.

➡➡ STATEMENT ————
 └ *required_choice1*
 required_choice2 ┘ ————— ➡➡

If choosing one of the items is optional, the entire stack appears below the main path.

➡➡ STATEMENT ————
 └ *optional_choice1*
 optional_choice2 ┘ ————— ➡➡

- ❑ If one of the items is the default, it appears above the main path, and the remaining choices are shown below it.

➡➡ STATEMENT ————
 └ *default_choice* ┘
 └ *optional_choice*
 optional_choice ┘ ————— ➡➡

- ❑ An arrow returning to the left above the main line indicates an item that can be repeated.

➡➡ STATEMENT ————
 └ *repeatable_item* ┘ ————— ➡➡

A repeat arrow above a stack indicates that the items in the stack can be repeated.

- ❑ Keywords appear in uppercase (for example, PARM1). They must be spelled exactly as shown but can be entered in lowercase. Variables appear in all lowercase letters (for example, *parmx*). They represent user-supplied names or values.
- ❑ If punctuation marks, parentheses, arithmetic operators, or such symbols are shown, they must be entered as part of the syntax.

Special Characters

(var) parameter of the CALL instruction 268
.environment 154, 265
.local 153, 265
.methods 265
.nil 265
.output 61
.rs 265

A

application assembly 179
ARG
 enhanced built-in function 275
ASCII user interface
 car dealer 57
 object 61
attribute 25, 26
audio 58, 118
 World Wide Web 207
AUI class 62

B

BLOB 114
 DB2 111, 243
 in Object REXX 114
 multimedia 118
 self-defining 121
Boxie the cat 113
browser
 class 34
 World Wide Web 184, 185
built-in functions
 enhanced
 ARG 275
 CONDITION 276
 DATATYPE 276
 DATE 277
 STREAM 277
 TIME 279
 new
 CHANGESTR 275
 COUNTSTR 276

C

CAE/OS2 134
CALL
 enhanced instruction 268
CALL instruction
 (var) parameter 268
car dealer
 application 37
 ASCII user interface 65
 billing GUI window 87
 class 159
 class relationships 146
 configurations 149
 customer view folder 175
 DB2 setup 250
 directories 254
 directory structure 147
 file structure 150
 GUI 81, 82
 home page 212
 installation 241
 main GUI window 83
 methods 43
 model 41
 multimedia files 292
 objects 40
 part list GUI window 84
 relationships 48
 remove installation 260
 run 253
 run on World Wide Web 254
 sample data 289
 service item list GUI window 85
 simple installation GUI window 156
SOM 162
SOM IR file 170
source code 287
 source code for base classes 293
 source code for DB2 persistence 317
 source code for file persistence 310
 source code for running the
 programs 327
 use case 38
 work orders GUI window 86
World Wide Web 212
WPS 174
WPS folders 174
CCD 112
CD
 content 241

- running sample applications 242
- CGI 183, 186, 192
 - environment variables 193
 - REXX interface 202, 203, 204
- CHANGESTR
 - new built-in function 275
- class 19
 - abstract 21
 - browser 34
 - communication 152
 - directive 19, 265
 - library 33
 - meta 30
 - mixin 77
 - persistence 77
 - structure 74
- Classy Cars 58, 82, 102, 111, 131
- client/server 134, 141
- CLOB 114
- collection classes 33, 43, 49
- Common Gateway Interface , See CGI
- component 14
- concurrency 4, 217
- CONDITION
 - enhanced built-in function 276
- condition traps
 - debug routine 280
 - SIGNAL and CALL 280
- CONFIG.SYS 245
- configuration
 - DB2 158
 - file system 158
- configuration file 148
- configuration management 143
- cookies
 - chocolate chip 63
- CORBA 14, 161
- COUNTSTR
 - new built-in function 276
- CRUD methods 43
- CUA 90

D

- DAP 38
- DATATYPE
 - enhanced built-in function 276
- DATE
 - enhanced built-in function 277
- DB2
 - authorization 130
 - BLOB 111, 114
 - database 104, 250

- implementation 108
- load 105, 252
- multimedia implementation 125
- persistence 101
- persistent methods 106
- prerequisites 243
- security 129
- setup 250
- stored procedures 133
- tables 251

- DBCLOB 114
- deadlock 229
- debugging
 - condition traps 280
- declaratives 17
- DEVCON 38, 90
- digital camera 112
- directives
 - class 19, 265
 - in Dr. Dialog 92
 - in GUI builders 92
 - in VisPro/REXX 93
 - in Watcom VX•REXX 94
 - method 19, 266
 - Object REXX 264, 265
 - requires 76, 148, 267
 - routine 53, 267
- DO
 - enhanced instruction 269
- do over 4, 269
- DOS 58
- Dr. Dialog
 - car dealer billing window 87
 - car dealer main window 83
 - car dealer part list window 84
 - car dealer service items window 85
 - car dealer simple installation window 156
 - car dealer work orders window 86
 - development environment 95
 - directives 92
 - GUI builder 5, 82
 - installation program window 244
 - philosopher's forks 227
 - sample application run window 253
- drag and drop 176
- DSOM 163
- dynamic SQL 130

E

- encapsulation 12, 25
- EXPOSE

new instruction 270
expose 25

F

FAT 67
file locator host variable 115
folder
 car dealer show 174
 drag and drop 114
 ObjectREXX Redbook 247
 philosopher's forks 248
FORWARD
 new instruction 270

G

global environment directory 154, 265
GUARD
 new instruction 271
guard 220, 226
GUI builder 82, 88
 development environment 95

H

Hacurs 37
 home page 187
home page 185, 187
 car dealer 212
 Hacurs 187
host variable
 file locator 116
HPFS 67, 69
HTML 183, 185
 class 194
 form 197
Hypertext Markup Language , See HTML

I

IDL 163, 167
inheritance 20
 car dealer 73
 meta class 30
 multiple 23
installation
 car dealer 241

Object REXX 242
remove sample applications 260
sample applications 242
installation program
 considerations 155
 redbook examples 243
instructions
 enhanced
 CALL 268
 DO 269
 PARSE 271
 SIGNAL 274
 new
 EXPOSE 270
 FORWARD 270
 GUARD 271
 RAISE 272
 REPLY 273
 USE 275
Internet , See World Wide Web
Internet Connection Server 185, 192
 administration file 201
ITSO xxviii

L

large objects , See BLOB
lazy write 69
load on demand 109
LOB 114
local directory 153, 265

M

menu 63
 data file 64
 loop 64
 object 63
 operations 63
message object 218
meta class 30, 168
method 18, 28
 class 29
 directive 19, 25, 266
 init 27
 instance 29
 new 27
 private 29
 unguarded 219
migration considerations 286
mixin 77

- multimedia 58, 118, 243
 - BLOB 111
 - files 292
 - World Wide Web 206

N

- new
 - method 27

O

- object
 - concurrency 217
 - cooperation 13
 - creation 27, 50
 - destruction 28, 50
 - file persistence 67
 - instance 27
 - instance management 51
 - message 218
 - methods 18
 - model 52
 - persistence 67
 - relationships 48
 - stream (file) format 71
- object management group 14
- Object REXX
 - class library 33
 - client/server 136
 - concurrency 217
 - configuration management 143
 - DB2 stored procedures 133
 - enhanced instructions 268
 - installation 242
 - migration 286
 - new features 263
 - new instructions 268
 - security 129
 - shared objects 134
 - SOM 161
 - switch to classic REXX 242
 - using BLOBs 114
 - using SOM object 169
 - why 15
 - World Wide Web 183
 - WPS 173
- object-oriented
 - benefits 6
 - languages 9
 - why 6

- ODBC 162
- OMG 14

P

- PARSE
 - enhanced instruction 271
- parse 4
- performance 203
- persistence
 - DB2 101
 - DB2 methods 106
 - file system 67
 - persistent class 77
- philosopher's forks 223
 - directory 260
 - GUI window 227
 - OS/2 window 226
- pictures
 - camera 112
 - World Wide Web 206
- polymorphism 30
- PPP 184
- prerequisites 243
- productivity 6
- propagate parameter of RAISE 273
- prototyping 9

R

- RAISE
 - new instruction 272
 - test of propagate parameter 273
- redbook examples
 - folder 247
 - installation program 243
 - run window 253
 - source code 287
- REPLY
 - early 218
 - example 220
 - new instruction 273
- requires
 - directive 4, 76, 148, 267
- reuse 7, 12
- REXXC utility 131, 264, 267
- routine
 - directive 53, 267

S

- sample applications
 - directories 254
 - installation 242
 - remove 260
 - run 253
- security 129
- self
 - variable 264
- server
 - World Wide Web 185
- SIGNAL
 - enhanced instruction 274
- signal 4
- SLIP 163, 184
- Smalltalk 6, 23
- SOM
 - class 168
 - IDL for Part 167
 - IDL for PartMeta 168
 - implementation steps 170
 - implementing an object 166
 - interface repository 170
 - object 4, 163, 167
 - object broker 15
 - Object REXX 16, 17, 30, 161
- source code
 - base classes 293
 - car dealer 287
 - car dealer run 327
 - DB2 persistence 317
 - file persistence 310
 - multimedia 292
 - sample data 289
- source code listings
 - CARCUST.CLS 293, 311, 317
 - CARDEAL.CLS 308
 - CARMODEL.CFG 310, 317
 - CARPART.CLS 316, 326
 - CAR-RUN.CMD 327
 - CARSERV.CLS 302, 315, 325
 - CARVEHI.CLS 296, 312, 319
 - CARWORK.CLS 297, 314, 322
 - CUSTOMER.DAT 289
 - MEDIA.DAT 292
 - PART.DAT 291
 - PART.ORI 304
 - PART.SOM 307
 - PERSIST.CLS 308
 - SERVICE.DAT 291
 - VEHICLE.DAT 290
 - WORKORD.DAT 290
- spiral method 8

- stem 33, 286
- stored procedures 133
- stream 286
- STREAM - enhanced built-in function 277
- subclass 20
- subdirectories
 - car dealer 147
- subroutines 4
- super
 - variable 264
- SWITCHRX 242
- syntax diagram descriptions 329

T

- tilde 18, 28
- TIME
 - enhanced built-in function 279
- Trusty Trucks 38, 67

U

- unguarded method 219
- USE
 - example 28
 - new instruction 275
- use case 39
- user interface
 - ASCII 57
 - GUI 81
- utilities
 - new 282

V

- Value Vans 162, 173
- VAR
 - new built-in function 279
- variable
 - methods and 43
 - pool 25
 - special 264
- video 58, 118
 - World Wide Web 207
- VisPro/REXX
 - development environment 97
 - directives 93
 - GUI builder 5, 90
 - philosopher's forks 235

VX•REXX , See Watcom VX•REXX

W

Warp 243

Warp Connect 58, 163

Watcom VX•REXX

- development environment 99

- directives 94

- GUI builder 5, 90

- philosopher's forks 236

waterfall method 8

Web , See World Wide Web

WebExplorer 185

Workplace Shell , See WPS

World Wide Web

- audio 207

- browser 184

- car dealer 212

- home page 185

- Internet Connection Server 185

- Internet name 186

- multimedia 206

- Object REXX 183

- pictures 206

- server 185

- video 207

- WebExplorer 185

WPS 16, 173, 282

ACKNOWLEDGMENT

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT, AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU ALSO AGREE THAT THIS AGREEMENT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN YOU AND THE COMPANY AND SUPERSEDES ALL PROPOSALS OR PRIOR AGREEMENTS, ORAL, OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN YOU AND THE COMPANY OR ANY REPRESENTATIVE OF THE COMPANY RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

Should you have any questions concerning this Agreement or if you wish to contact the Company for any reason, please contact in writing at the address below.

Robin Short
Prentice Hall PTR
One Lake Street
Upper Saddle River, New Jersey 07458

IBM OBJECT REXX FOR OS/2 EVALUATION LICENSE

BEFORE USING THIS PROGRAM YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS. USING THE PROGRAM INDICATES YOUR ACCEPTANCE OF THE FOLLOWING TERMS AND CONDITIONS.

IBM Object REXX for OS/2 ("Program") is copyrighted and licensed, not sold. The term "Program" means the original and all whole or partial copies of it. International Business Machines Corporation or one of its subsidiaries ("IBM") owns or has licensed from the owner copyrights in the Program.

IBM grants you free of charge a non-exclusive, nontransferable license to download the Program and use it to enable you to evaluate the potential usefulness of the Program to you.

You may copy the Program for back-up purposes only. You may not distribute, sublicense, lease or rent the Program, or create derivative works, reverse compile, reverse assemble or otherwise attempt to translate or seek to gain access to the Program's source code, except as permitted by law without the possibility of contractual waiver.

The term of your License will be one year from your first use. You must destroy and/or delete the Program within ten (10) days after the expiration of this one year period.

You may terminate this license at any time. IBM may terminate this license at any time if you are in breach of any of its terms. In either event, you must destroy all copies of the Program.

IBM IS PROVIDING THE PROGRAM TO YOU "AS IS", WITHOUT ANY WARRANTIES (EXPRESS OR IMPLIED) OR SUPPORT WHATSOEVER, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE.

IN NO EVENT WILL IBM BE LIABLE FOR ANY DAMAGE WHATSOEVER, INCLUDING, BUT NOT LIMITED TO, LOST PROFITS, LOST SAVINGS, INCIDENTAL OR INDIRECT DAMAGES OR OTHER ECONOMIC CONSEQUENTIAL DAMAGES, EVEN IF IBM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN ADDITION, IBM WILL NOT BE LIABLE FOR ANY DAMAGES CLAIMED BY YOU BASED ON ANY THIRD-PARTY CLAIM.

The above limitation of remedies also applies to any developer and/or supplier of the Program. Such developer and/or supplier is an intended beneficiary of these limitation terms.

As your testing and evaluation of this Program is important to IBM, this license is granted without charge. You agree that IBM may use all suggestions, improvements and written materials you may furnish IBM in connection with your use of this Program, and that IBM may include them in any IBM product without accounting to you.

If you acquire the Program in the United States, this license is governed by the laws of the State of New York. If you acquire the Program in Canada, this license is governed by the laws of the Province of Ontario. Otherwise, the license is governed by the laws of the country in which you acquire the Program.

Note to U.S. Government Users -- Documentation related to Restricted Rights -- Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corporation.

LICENSE AGREEMENT AND LIMITED WARRANTY

READ THE FOLLOWING TERMS AND CONDITIONS CAREFULLY BEFORE OPENING THIS CD PACKAGE. THIS LEGAL DOCUMENT IS AN AGREEMENT BETWEEN YOU AND PRENTICE-HALL, INC. (THE "COMPANY"). BY OPENING THIS SEALED CD PACKAGE, YOU ARE AGREEING TO BE BOUND BY THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THESE TERMS AND CONDITIONS, DO NOT OPEN THE CD PACKAGE. PROMPTLY RETURN THE UNOPENED CD PACKAGE AND ALL ACCOMPANYING ITEMS TO THE PLACE YOU OBTAINED THEM FOR A FULL REFUND OF ANY SUMS YOU HAVE PAID.

1. **GRANT OF LICENSE:** In consideration of your purchase of this book, and your agreement to abide by the terms and conditions of this Agreement, the Company grants to you a nonexclusive right to use and display the copy of the enclosed software program (hereinafter the "SOFTWARE") on a single computer (i.e., with a single CPU) at a single location so long as you comply with the terms of this Agreement. The Company reserves all rights not expressly granted to you under this Agreement.
2. **OWNERSHIP OF SOFTWARE:** You own only the magnetic or physical media (the enclosed CD) on which the SOFTWARE is recorded or fixed, but the Company and the software developers retain all the rights, title, and ownership to the SOFTWARE recorded on the original CD copy(ies) and all subsequent copies of the SOFTWARE, regardless of the form or media on which the original or other copies may exist. This license is not a sale of the original SOFTWARE or any copy to you.
3. **COPY RESTRICTIONS:** This SOFTWARE and the accompanying printed materials and user manual (the "Documentation") are the subject of copyright. The individual programs on the CD are copyrighted by the authors of each program. Some of the programs on the CD include separate licensing agreements. If you intend to use one of these programs, you must read and follow its accompanying license agreement. You may not copy the Documentation or the SOFTWARE, except that you may make a single copy of the SOFTWARE for backup or archival purposes only. You may be held legally responsible for any copying or copyright infringement which is caused or encouraged by your failure to abide by the terms of this restriction.
4. **USE RESTRICTIONS:** You may not network the SOFTWARE or otherwise use it on more than one computer or computer terminal at the same time. You may physically transfer the SOFTWARE from one computer to another provided that the SOFTWARE is used on only one computer at a time. You may not distribute copies of the SOFTWARE or Documentation to others. You may not reverse engineer, disassemble, decompile, modify, adapt, translate, or create derivative works based on the SOFTWARE or the Documentation without the prior written consent of the Company.
5. **TRANSFER RESTRICTIONS:** The enclosed SOFTWARE is licensed only to you and may not be transferred to any one else without the prior written consent of the Company. Any unauthorized transfer of the SOFTWARE shall result in the immediate termination of this Agreement.
6. **TERMINATION:** This license is effective until terminated. This license will terminate automatically without notice from the Company and become null and void if you fail to comply with any provisions or limitations of this license. Upon termination, you shall destroy the Documentation and all copies of the SOFTWARE. All provisions of this Agreement as to warranties, limitation of liability, remedies or damages, and our ownership rights shall survive termination.
7. **MISCELLANEOUS:** This Agreement shall be construed in accordance with the laws of the United States of America and the State of New York and shall benefit the Company, its affiliates, and assignees.
8. **LIMITED WARRANTY AND DISCLAIMER OF WARRANTY:** The Company warrants that the SOFTWARE, when properly used in accordance with the Documentation, will operate in substantial conformity with the description of the SOFTWARE set forth in the Documentation. The Company does not warrant that the SOFTWARE will meet your requirements or that the operation of the SOFTWARE will be uninterrupted or error-free. The Company warrants that the media on which the SOFTWARE is delivered shall be free from defects in materials and workmanship under normal use for a period of thirty (30) days from the date of your purchase. Your only remedy and the Company's only obligation under these limited warranties is, at the Company's option, return of the warranted item for a refund of any amounts paid by you or replacement of the item. Any replacement of SOFTWARE or media under the warranties shall not extend the original warranty period. The limited warranty set forth above shall not apply to any SOFTWARE which the Company determines in good faith has been subject to misuse, neglect, improper installation, repair, alteration, or damage by you.

EXCEPT FOR THE EXPRESSED WARRANTIES SET FORTH ABOVE, THE COMPANY DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. EXCEPT FOR THE EXPRESS WARRANTY SET FORTH ABOVE, THE COMPANY DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATION REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE.

IN NO EVENT, SHALL THE COMPANY OR ITS EMPLOYEES, AGENTS, SUPPLIERS, OR CONTRACTORS BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR IN CONNECTION WITH THE LICENSE GRANTED UNDER THIS AGREEMENT, OR FOR LOSS OF USE, LOSS OF DATA, LOSS OF INCOME OR PROFIT, OR OTHER LOSSES, SUSTAINED AS A RESULT OF INJURY TO ANY PERSON, OR LOSS OF OR DAMAGE TO PROPERTY, OR CLAIMS OF THIRD PARTIES, EVEN IF THE COMPANY OR AN AUTHORIZED REPRESENTATIVE OF THE COMPANY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL LIABILITY OF THE COMPANY FOR DAMAGES WITH RESPECT TO THE SOFTWARE EXCEED THE AMOUNTS ACTUALLY PAID BY YOU, IF ANY, FOR THE SOFTWARE.

SOME JURISDICTIONS DO NOT ALLOW THE LIMITATION OF IMPLIED WARRANTIES OR LIABILITY FOR INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATIONS MAY NOT ALWAYS APPLY. THE WARRANTIES IN THIS AGREEMENT GIVE YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY IN ACCORDANCE WITH LOCAL LAW.



Object REXX for OS/2 Warp

Trevor Turton ♦ Ueli Wahli

Object orientation (OO) is a topic of great interest and concern today. Some developers who use OO claim that it significantly increases productivity; others view it as good for rocket science, not for business.

Many OO languages seem complicated and alien to programmers familiar with procedural languages such as COBOL. This book introduces Object REXX, a new OO language that breaks the OO barrier. Object REXX is based on a tried-and-trusted language used around the world today. Because it has the most complete and easy-to-use set of OO features of any language, it offers a simple way for programmers with a procedural background to enter the new world of objects. Object REXX also supports distributed objects written in many other languages through Common Object Request Broker Architecture (CORBA) technologies, such as IBM's System Object Model (SOM).

This book demonstrates a practical approach to using Object REXX and OO techniques to develop commercial systems to meet changing business requirements. It tells the story of how Hanna, Steve, and Curt design and implement a commercial application system step by step, using object persistence in file systems and relational databases, GUI builders, existing SOM-based objects, the OS/2 Workplace Shell, and Internet Web pages. Extensive code examples are provided to illustrate every step.

Sharing Technical Expertise from around the World. This book and other IBM Redbooks are products of IBM's International Technical Support Organization, where worldwide specialists work alongside you to harness IBM technologies. IBM Redbooks make the answers to your most pressing technical questions easily and immediately accessible.

For more information: <http://www.redbooks.ibm.com/redbooks>

PRENTICE HALL
Upper Saddle River, NJ 07458
<http://www.prenhall.com>

U.S. \$38.95
Canada \$54.95

PRICE \$38.95

OBJECT REXX FOR OS/2 WARP/CD



WAHLI U 3598 -OTHER PRO LAN II 151
1629747 DR 1 011397 PRNT 0132734672



ISBN 0-13-273467-2

